

c o n f e r e n c e

*proceedings*

# 2000 USENIX Annual Technical Conference

*San Diego, CA, USA  
June 18–23, 2000*

Sponsored by

**USENIX**  
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

USENIX Proceedings of the 2000 USENIX Annual Technical Conference

San Diego, CA, USA, June 2000



For additional copies of these proceedings contact:

USENIX Association  
2560 Ninth Street, Suite 215  
Berkeley, CA 94710 USA  
Phone: 510 528 8649  
FAX: 510 548 5738  
Email: office@usenix.org  
WWW URL: <http://www.usenix.org>

The price is \$32 for members and \$40 for nonmembers.

Outside the U.S.A. and Canada, please add  
\$18 per copy for postage (via air printed matter).

### **Past USENIX Technical Conferences**

1999 Monterey CA	1989 Summer Baltimore
1998 New Orleans	1989 Winter San Diego
1997 Anaheim	1988 Summer San Francisco
1996 San Diego	1988 Winter Dallas
1995 New Orleans	1987 Summer Phoenix
1994 Summer Boston	1987 Winter Washington, D.C.
1994 Winter San Francisco	1986 Summer Atlanta
1993 Summer Cincinnati	1986 Winter Denver
1993 Winter San Diego	1985 Summer Portland
1992 Summer San Antonio	1985 Winter Dallas
1992 Winter San Francisco	1984 Summer Salt Lake City
1991 Summer Nashville	1984 Winter Washington, D.C.
1991 Winter Dallas	1983 Summer Toronto
1990 Summer Anaheim	1983 Winter San Diego
1990 Winter Washington, D.C.	

© 2000 by The USENIX Association  
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-880446-22-7

Printed in the United States of America on 50% recycled paper, 10–15% post consumer waste.

JEFF MOGUL

**USENIX Association**

**Proceedings of the  
2000 USENIX  
Annual Technical Conference**

**June 18–23, 2000  
San Diego, California, USA**

## Conference Organizers

### Program Chair

Christopher Small, *Osprey Partners LLC*

### Program Committee

Ken Arnold, *Sun Microsystems*

Aaron Brown, *University of California at Berkeley*

Fred Douglass, *AT&T Labs—Research*

Edward W. Felten, *Princeton University*

Eran Gabber, *Bell Labs Research, Lucent Technologies*

Greg Minshall, *Redback Networks, Inc.*

Yoonho Park, *IBM Research*

Vern Paxson, *ACIRI*

Liuba Shrira, *Brandeis University*

Keith A. Smith, *Harvard University*

Mark Zbikowski, *Microsoft*

### Invited Talks Coordinators

John Heidemann, *USC/Information Sciences Institute*

John T. Kohl, *Rational Software*

### The USENIX Association Staff

## External Reviewers

Sharad Agarwal

Charles A. Antonelli

Mohit Aron

Ricardo Bianchini

David Black

Trevor Blackwell

Daniel Bleichenbacher

Steve Blott

José Brustoloni

Ramón Cáceres

Enrique V. Carrera-Erazo

Yih-Farn Chen

Tzi-cker Chiueh

Crispin Cowan

Steven Gribble

Yennun Huang

Galen Hunt

Liviu Iftode

Ben Jai

Brian Kernighan

Terence Kelly

Robin Kravetz

Geoff Kuenning

Marshall Kirk McKusick

Richard Martin

Wee Teck Ng

Thu Nguyen

Stephen North

Michael Ogg

David Oppenheimer

David Oran

Banu Özden

Shirish Phatak

Rob Pike

Drew Roselli

Brian Schmidt

Margo Seltzer

Jonathan Shapiro

Elizabeth Shriver

Sandeep Sibal

Oliver Spatscheck

Christopher Stein

Dave Sullivan

Randi Thomas

Brett Vickers

Bulent Yener

Cliff Young

Erez Zadok

# 2000 USENIX Annual Technical Conference

June 18–23, 2000  
San Diego, California, USA

Index of Authors .....	vii
Message from the Program Chair .....	ix

## Wednesday, June 21

### Instrumentation and Visualization

Session Chair: Christopher Small, Osprey Partners LLC

Mapping and Visualizing the Internet .....	1
<i>Bill Cheswick, Bell Laboratories; Hal Burch, Carnegie Mellon University; Steve Branigan, Bell Laboratories</i>	
Measuring and Characterizing System Behavior Using Kernel-Level Event Logging .....	13
<i>Karim Yaghmour and Michel R. Dagenais, Ecole Polytechnique de Montréal</i>	
Pandora: A Flexible Network Monitoring Platform .....	27
<i>Simon Patarin and Mesaac Makpangou, INRIA SOR Group, Rocquencourt</i>	

### File Systems

Session Chair: Liuba Shrira, Brandeis University

A Comparison of File System Workloads .....	41
<i>Drew Roselli and Jacob R. Lorch, University of California at Berkeley; Thomas E. Anderson, University of Washington</i>	
FiST: A Language for Stackable File Systems .....	55
<i>Erez Zadok and Jason Nieh, Columbia University</i>	
Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems .....	71
<i>Margo I. Seltzer, Harvard University; Gregory R. Ganger, Carnegie Mellon University; M. Kirk McKusick, Author &amp; Consultant; Keith A. Smith, Harvard University; Craig A. N. Soules, Carnegie Mellon University; Christopher A. Stein, Harvard University</i>	

### Old Dogs, New Tricks

Session Chair: Greg Minshall, Redback Networks, Inc.

Lexical File Names in Plan 9, or, Getting Dot-Dot Right .....	85
<i>Rob Pike, Bell Laboratories</i>	
Gecko: Tracking a Very Large Billing System .....	93
<i>Andrew Hume, AT&amp;T Labs—Research; Scott Daniels, Electronic Data Systems Corp.; Angus MacLellan, AT&amp;T Labs—Research</i>	
Extended Data Formatting Using Sflo .....	107
<i>Glenn S. Fowler, David G. Korn, Kiem-Phong Vo, AT&amp;T Labs—Research</i>	



## Thursday, June 22

### Distribution and Scalability: Problems and Solutions

*Session Chair: Ken Arnold, Sun Microsystems*

Virtual Services: A New Abstraction for Server Consolidation .....117

*John Reumann, University of Michigan; Ashish Mehra, IBM T.J. Watson Research Center; Kang G. Shin, University of Michigan; Dilip Kandlur, IBM T.J. Watson Research Center*

Location-Aware Scheduling with Minimal Infrastructure .....131

*John Heidemann, USC/ISI; Dhaval Shah, Noika*

Distributed Computing: Moving from CGI to CORBA .....139

*James FitzGibbon and Tim Strike, Targetnet.com Inc.*

### Tools

*Session Chair: Eran Gabber, Lucent Technologies—Bell Labs*

Outwit: UNIX Tool-Based Programming Meets the Windows World .....149

*Diomidis D. Spinellis, University of the Aegean*

Plumbing and Other Utilities .....159

*Rob Pike, Bell Laboratories*

Integrating a Command Shell into a Web Browser .....171

*Robert C. Miller and Brad A. Myers, Carnegie Mellon University*

### Kernel Structures

*Session Chair: Keith A. Smith, Harvard University*

Operating System Support for Multi-User, Remote, Graphical Interaction .....183

*Alexander Ya-li Wong and Margo I. Seltzer, Harvard University*

Techniques for the Design of Java Operating Systems .....197

*Godmar Back, Patrick Tullmann, Leigh Stoller, Wilson C. Hsieh, and Jay Lepreau, University of Utah*

Signaled Receiver Processing .....211

*José Brustoloni, Eran Gabber, Abraham Silberschatz, and Amit Singh, Lucent Technologies—Bell Laboratories*

## Friday, June 23

### Run-Time Tools and Tricks

*Session Chair: Christopher Small, Osprey Partners LLC*

DITools: Application-level Support for Dynamic Extension and Flexible Composition .....225

*Albert Serra, Nacho Navarro, and Toni Cortes, Universitat Politècnica de Catalunya*

Portable Multithreading—The Signal Stack Trick for User-Space Thread Creation .....239

*Ralf S. Engelschall, Technische Universität München (TUM)*

Transparent Run-Time Defense Against Stack-Smashing Attacks .....251

*Arash Baratloo and Navjot Singh, Bell Labs Research, Lucent Technologies; Timothy Tsai, Reliable Software Technologies*

## **Measurement and Stability**

*Session Chair: Fred Douglass, AT&T Labs—Research*

Towards Availability Benchmarks: A Case Study of Software RAID Systems .....263  
*Aaron Brown and David A. Patterson, University of California at Berkeley*

Performing Replacement in Modem Pools .....277  
*Yannis Smaragdakis, Georgia Institute of Technology; Paul Wilson, University of Texas at Austin*

Auto-Diagnosis of Field Problems in an Appliance Operating System .....293  
*Gaurav Banga, Network Appliance, Inc.*

## **Servers: Load Balancing and Scheduling**

*Session Chair: Yoonho Park, IBM Research*

Dynamic Function Placement for Data-Intensive Cluster Computing .....307  
*Khalil Amiri, David Petrou, Gregory R. Ganger, and Garth A. Gibson, Carnegie Mellon University*

Scalable Content-aware Request Distribution in Cluster-based Network Servers .....323  
*Mohit Aron, Darren Sanders, Peter Druschel, and Willy Zwaenepoel, Rice University*

Isolation with Flexibility: A Resource Management Framework for Central Servers .....337  
*David G. Sullivan and Margo I. Seltzer, Harvard University*



## Index of Authors

Amiri, Khalil . . . . .	307	Myers, Brad A. . . . .	171
Anderson, Thomas E. . . . .	41	Navarro, Nacho . . . . .	225
Aron, Mohit . . . . .	323	Nieh, Jason . . . . .	55
Back, Godmar . . . . .	197	Patarin, Simon . . . . .	27
Banga, Gaurav . . . . .	293	Patterson, David A. . . . .	263
Baratloo, Arash . . . . .	251	Petrou, David . . . . .	307
Branigan, Steve . . . . .	1	Pike, Rob . . . . .	85, 159
Brown, Aaron . . . . .	263	Reumann, John . . . . .	117
Brustoloni, José . . . . .	211	Roselli, Drew . . . . .	41
Burch, Hal . . . . .	1	Sanders, Darren . . . . .	323
Cheswick, Bill . . . . .	1	Seltzer, Margo I. . . . .	71, 183, 337
Cortes, Toni . . . . .	225	Serra, Albert . . . . .	225
Dagenais, Michel R. . . . .	13	Shah, Dhaval . . . . .	131
Daniels, Scott . . . . .	93	Shin, Kang G. . . . .	117
Druschel, Peter . . . . .	323	Silberschatz, Abraham . . . . .	211
Engelschall, Ralf S. . . . .	239	Singh, Amit . . . . .	211
FitzGibbon, James . . . . .	139	Singh, Navjot . . . . .	251
Fowler, Glenn S. . . . .	107	Smaragdakis, Yannis . . . . .	277
Gabber, Eran . . . . .	211	Smith, Keith A. . . . .	71
Ganger, Gregory R. . . . .	71, 307	Soules, Craig A. N. . . . .	71
Gibson, Garth A. . . . .	307	Spinellis, Diomidis D. . . . .	149
Heidemann, John . . . . .	131	Stein, Christopher A. . . . .	71
Hsieh, Wilson C. . . . .	197	Stoller, Leigh . . . . .	197
Hume, Andrew . . . . .	93	Strike, Tim . . . . .	139
Kandlur, Dilip . . . . .	117	Sullivan, David G. . . . .	337
Korn, David G. . . . .	107	Tsai, Timothy . . . . .	251
Lepreau, Jay . . . . .	197	Tullmann, Patrick . . . . .	197
Lorch, Jacob R. . . . .	41	Vo, Kiem-Phong . . . . .	107
McKusick, M. Kirk . . . . .	71	Wilson, Paul . . . . .	277
MacLellan, Angus . . . . .	93	Wong, Alexander Ya-li . . . . .	183
Makpangou, Mesaac . . . . .	27	Yaghmour, Karim . . . . .	13
Mehra, Ashish . . . . .	117	Zadok, Erez . . . . .	55
Miller, Robert C. . . . .	171	Zwaenepoel, Willy . . . . .	323





# Message from the Conference Chair

Welcome to San Diego and the 2000 USENIX Annual Technical Conference!

2000 is the 25th anniversary of USENIX and the 30th anniversary of UNIX. (And, depending on how you count, this is either the last USENIX of the 20th century or the first of the 21st—either way, it's a cause for celebration.)

Kirk McKusick has done an exemplary job organizing the FREENIX Track, and John Heidemann and John Kohl have put together what looks like the best slate of invited talks we've had in years. The tutorial program, organized by Dan Klein, is, as always, outstanding.

We have an excellent technical program this year. We received 90 submissions for the refereed track, of which we accepted 27. The program committee members were very happy with both the number and the quality of the submissions, which made our job more difficult but, in the end, more rewarding. Over the course of five weeks (and the Christmas/New Year/Y2K holiday), the program committee and 48 external reviewers wrote a total of 370 reviews, slightly more than 4 reviews per paper. The program committee then met for two days to discuss the papers, decide which to accept, assign papers to sessions, and schedule the sessions over the course of the conference.

I greatly appreciate the program committee braving a snowstorm to attend the program committee meeting. This was the second year in a row that the meeting was held in New Jersey, in January, and the second year in a row we got hit with a freak snowstorm. (I wouldn't be surprised if this were the last time the USENIX Board selects someone from New Jersey to chair the conference!) Fred Douglass, who is both a past program chair and a member of this year's program committee, was the unofficial "vice-chair" of the program committee, acting as both a sounding board and a continuous source of helpful suggestions.

I thank the external reviewers for their efforts, especially those who reviewed six or more papers: Charles Antonelli, Liviu Iftode, Terence Kelly, Geoff Kuenning, Wee Teck Ng, Jonathan Shapiro, Oliver Spatscheck, and Erez Zadok. In addition, I would like to single out Liviu Iftode for special thanks; his efforts to raise the technical program to the highest possible standard were inspirational and went above and beyond the call of duty.

The USENIX staff, especially Jane-Ellen Long, Ellie Young, Judy DesHarnais, and Toni Veglia, make being a program chair as easy as it could possibly be. They were always willing to (gently) remind me when I was about to miss a deadline, took care of all of the heavy lifting of putting the conference together, and left me with the fun part of the job—organizing the technical program.

Finally, I would like to thank my former and current employers, Lucent Technologies—Bell Labs Research and Osprey Partners, and my former and current bosses, Avi Silberschatz and Jay Whipple, for their patience, understanding, and support.

Christopher Small, Program Chair



# Mapping and Visualizing the Internet

Bill Cheswick  
Bell Laboratories  
ches@bell-labs.com

Hal Burch\*  
Carnegie Mellon University  
hburch@cs.cmu.edu

Steve Branigan  
Bell Laboratories  
sbranigan@bell-labs.com

## Abstract

We have been collecting and recording routing paths from a test host to each of over 90,000 registered networks on the Internet since August 1998. The resulting database contains interesting routing and reachability information, and is available to the public for research purposes. The daily scans cover approximately a tenth of the networks on the Internet, with a full scan run roughly once a month. We have also been collecting Lucent's intranet data, and applied these tools to understanding its size and connectivity. We have also detected the loss of power to routers in Yugoslavia as the result of NATO bombing.

A simulated spring-force algorithm lays out the graphs that results from these databases. This algorithm is well known, but has never been applied to such a large problem. The Internet graph, with around 88,000 nodes and 100,000 edges, is larger than those previously considered tractable by the data visualization community. The resulting Internet layouts are pleasant, though rather cluttered. On smaller networks, like Lucent's intranet, the layouts present the data in a useful way. For the Internet data, we have also tried plotting a minimum distance spanning tree; by throwing away edges, the remaining graph can be made more accessible.

Once a layout is chosen, it can be colored in various ways to show network-relevant data, such as IP address, domain information, location, ISPs, and result of scan (completed, filtered, loop, etc).

This paper expands and updates the description of the project given in an IEEE Computer article [1].

## 1 Introduction

Network administrators have long used Van Jacobson's *traceroute* [15] to identify the path taken by outgoing packets towards a given destination. Each "hop" on the outgoing path is a router, and most

routers will respond to a *traceroute*-style packet with the IP address of one of its network interfaces.

By obtaining a list of all announced networks on an internet, and discovering the path to each of these networks, we build a good picture of the "center" of the Internet, and a kind of picture of what the Internet looks like as a whole. Of course, this is an egocentric view, as it only captures the paths taken by our outgoing packets. Thus, the picture is a reachability graph, not a complete map.

In the course of developing and testing our mapping software, we discovered that mapping is a more generally useful pursuit, as it became obvious that mapping an intranet is valuable. Large intranets are hard to manage and offer many security problems. A map can yield a lot of information and can help spot likely leaks in a company's perimeter security.

Each morning the mapping program scans two separate networks: Lucent's intranet and the Internet itself. On Lucent's intranet, the mapping program does run full scans daily. On the Internet, our daily scans cover about one tenth of the destinations, reaching each announced network about three times a month. The mapping program runs full scans of the Internet about once a month. The Internet data is published on a web page [23] and saved to CD-ROM. We plan to run these scans for years.

This scanning allows us to detect long-term routing and connectivity changes on the Internet. We are likely to miss the outage of a major backbone for a few hours, unless it happens while we are scanning. But a natural disaster, or major act of terrorism or war, may well show up.

Due to the magnitude of the resulting databases, a method of visualizing it is required. The eye can help us gain some understanding of the collected data. We can pick out interesting features for further investigation and find errors in Internet router configurations, such as routers that return invalid IP addresses. We'd like to have a large paper map with the properties of traditional flat maps: they can help one navigate towards destinations, determine connectivity, readily reveal major features and

\*Partially funded by an NSF Graduate Research Fellowship



interesting relationships, and are hard to fold up.

We use a spring-force algorithm to position the nodes on the map. A few simple rules govern the adjustment of a point's position based on proximity of graph neighbors, number of incident edges, and the number and position of close nodes that are not neighbors. We shuffle these points for 20 hours on a 400MHz Pentium to obtain the maps shown in this paper. The maps of Lucent take 20 minutes to an hour to lay out, depending on whether all the links are shown or just a spanning tree. Sample graph sizes are:

	networks	edges	nodes
Lucent	3,366	1,963	1,660
Internet	94,046	99,664	88,107

## 2 Motivation

The initial motivation for collecting path data came out of a Highlands Forum, a meeting that discussed possible responses to future infrastructure attacks using a scenario from the Rand Corporation. It was clear that a knowledge of the Internet's topology might be useful to law enforcement when the nation's infrastructure is under attack. Internet topology could also be useful for tracking anonymous packets back to their source [2].

An openly available map could be useful to monitor the connectivity of the Internet, and would be helpful to a variety of investigators. In particular, it might be useful to know how connectivity changes before and during an attack on the Internet infrastructure.

Good ISPs already watch this kind of information in near real-time to monitor the health of their own networks, but they rarely know anything (or care much) about the status of networks that are not directly connect to theirs. No one is responsible for watching the whole Internet. Of course, given its size, the entire Internet is difficult to watch. There is a major web of interconnecting ISPs that in some sense defines the "middle" of the net—the most important part.

Our current attempts, using *traceroute*-style packets, only map outgoing paths, and only from our test host—we discuss these limitations below. Even this limited connectivity information can yield insights about who is connected to whom.

The database itself can be useful for routing studies and graph theorists looking for real-world data to work with. Since we are collecting the data daily over a long period of time, we may be able to extract interesting trends. We systematically collect data daily, building a consistent database that can

be used to reconstruct routing on the Internet approximately for any day where mapping was done, at least the paths from our scanning host.

The mapping software has lent itself to another pressing problem: controlling an intranet. Software that can handle 100,000 nodes on the Internet can easily handle intranets of similar size. An intranet map can be colored to show insecure areas, business units, connections to remote offices, etc.

Our visualizations of the Internet itself have attracted wide media interest [25] [26]. Media generally visually represents the Internet by showing people staring at a web browser. Our maps give some idea of the size and complexity of the Internet.

## 3 Network Mapping

Our tracing data consists of paths from a test host towards a single host on a destination network. The list of possible destinations is obtained from the routing arbiter database [28]. This is a central registry of all assigned Internet addresses, including those used only privately. Each provides a target network address, such as 135.104.0.0/16.

We should also include networks announced in the core routing tables but not contained in the routing arbiter database. Preliminary analysis of these tables reveal that we miss approximately twenty percent of the networks. These omissions will be corrected when we start the multiple-source mapping described below.

We need to scan towards a particular host on the target network. It is not particularly important that the host actually be present. The network scanner randomly picks an IP number on each network that is likely to be in use. This random selection is biased based on a quick survey of commonly-used IP addresses (e.g., the most common last octet is 1 and lower numbers are more common). Essentially, we are performing a slow host scan over time until a responsive host is found.

If the trace reaches a host on the target network, the address is saved for future traces. More than half the traces end with silence (due to an invalid address or firewall) or an ICMP error reporting failure.

This technique only records an outgoing packet path. The incoming path is often different: many Internet routes are asymmetric, as ISP interconnect agreements often divert traffic through different connections. We do not know of a reliable way to discover return packet paths, but some ideas are discussed in section 7.

The path may vary between traces, or even individual probes, depending on outages, redundant

links, reconfigurations, etc. This means the mapping program may occasionally ‘discover’ paths that don’t exist. Imagine a packet to Germany that is either routed through the United Kingdom or France at random, for example. As alternate packets travel through alternate paths, the mapping program will infer connections between the alternate paths that do not exist. We believe that load-balancing over large stretches of paths is rare, so the effect of them is limited. In terms of outages and routing changes, the number of routes changing during a scan should be relatively small in most cases.

The technique employed only discovers the IP path. Each link along this path may not actually represent a physical link. For example, if an ISP is running their backbone over ATM, then each link represents a virtual circuit that may travel through many ATM nodes. Depending on how the ATM network is configured, such an ISP’s backbone may appear to be completely connected, even though it isn’t physically true. From an IP standpoint, however, detecting the physical connectivity is extremely difficult.

The target, date, path data, and path completion codes are recorded in a simple text format, described in appendix A. The database is manipulated with traditional UNIX text tools and some simple additional programs.

Each day’s database is compressed and stored permanently. Copies are available upon request. The latest Internet database is available daily online [23]. The compressed database is about 10–20MB: we periodically strip out old paths in order to keep its size down (Special snapshots of the database are taken before this, however).

### 3.1 Mapping, Not Hacking

We do not want our tracing to be confused with hacking probes, so the mapping must proceed gingerly. The mapping program probes with UDP packets addressed to high port numbers ranging from about 33,000 to 50,000. Most intrusion detection systems recognize these as *traceroute*-style packets, though our port range is larger than *traceroute*’s. At worst, the probes tend to confuse system administrators, as there are few real services that use these ports.

The path is discovered one hop at a time. For each hop, a probe is sent out. If no reply is received in 5 seconds, a second probe is sent. If no reply is received to the second probe in 15 more seconds, a third probe is sent. If no reply is received within 45 seconds after the third probe is sent, the path dis-

covery is halted. Stopping a path trace after failing only one hop stops us from discovering the second half of many paths [5], but makes us less threatening network citizens. A new scanner will try one hop beyond these IP “holes”, giving us some idea of what we are missing.

Since we do not want our mapping to be confused with hacking network probes, it is vital that curious system administrators can easily determine what we are doing. Our first clue to them is the name of our mapping host, **ches-netmapper**, and the domain **research.bell-labs.com**. This name itself tells most of the story, and we think this makes most administrators who do notice the packets nod and move on to other work.

We maintain a web page describing this project [22]. Tom Limoncelli, who runs the network that contains our mapping host, has had to field a number of queries about our activities, added a DNS TXT record to **netmapper**’s entry that points to our web page. In addition, he suggested the world’s shortest (and safest) web server to direct queries to the project’s web page (the web server just **cat**’s a file).

A few network administrators have complained. They either did not like the probe, or our packets cluttered their logs. (The Australian Parliament was the first on the list!). We record these networks in an opt-out list and cease probing them. Certainly others may have simply blocked our packets, or filtered our probes out of their logs. It would be interesting to compare hosts that were reached early in the scans and later fell out of sight.

We have been in touch with a number of emergency response groups to explain our activity. We want them to understand the mapping activity and satisfy their justifiable curiosity. We would have a much harder time justifying our probes if we ran overt host or port scans, which often precede a hacking attack. We believe only a tiny percentage of the Internet system administrators have noticed our mapping efforts.

The mapping machine itself is highly resistant to network invasion: some other network scans have promoted powerful hacking responses. Of course, like any other publicly-accessible machine, it could fall to denial-of-service attacks.

## 4 Map Layout

We use a force-directed method similar to previous work [8] [6] to lay out the graph. The basic idea is to model the graph as a physical system and then to find the set of node positions that minimizes the total energy. The standard model employed is

spring attraction and electrical repulsion. Attraction is done by connecting any two connected nodes by a spring. The repulsive force derives by giving each node a positive electrical charge, so that they repel each other.

Once you have this model, finding a minimum has been well studied. In particular, the most common techniques are gradient descent [13], conjugate gradient [14], and simulated annealing [13]. None of these algorithms are guaranteed to find a global minimum in a reasonable time, but they are able to approximately minimum configurations. We choose to use gradient descent because of the ease of coding it.

Previous work on graph drawing, however, has considered graphs the size of our Internet dataset as huge [9] [16], and extending the runtime results of previous work to our graph and adjusting for a faster machine yield times on the order of months to millennia. Thus, the standard algorithms are too slow for our graph. We employ two tricks to more quickly compute a layout, at the cost of possibly being less optimal.

The first trick is replace the electrical repulsive field with spring repulsion. Imagine that any two nodes which do not share an edge are connected, via infinite strings, to a spring. Thus, if the nodes are further apart than the rest length of the spring, there is no force applied. If the nodes are closer than the spring's rest length, the spring is compressed, and the nodes are pushed apart. This gives us a bounded repulsive force, which means that instead of having to calculate a quadratic number of repulsive forces, the number of repulsive forces goes down to approximately linear, since pairs of points that are further apart than the rest length can be ignored. Thus, the exact repulsive force on each node can be calculated in approximately linear time.

The real optimization, however, is laying out the graph one layer at a time. First the links to our three ISPs are laid out and the system is iterated until they stop moving "very much." Then, all the routers one hop further away are added, and the system is iterated (which may move the nodes from previous levels as well). Then the next hop, and so on. This tends to give placement based on information high in the tree. A movie of an early version of the layout process for Lucent data is available at our web page [24].

Our original layouts showed all the paths. This resulted in a picture such as figure 1<sup>1</sup>. While the

<sup>1</sup>Due to printing limitations, all figures are rendered in black and white. To view color figures, visit our web site at <http://www.cheswick.com/ches/map/mapfigs/>

middle is mostly a muddle, the edges showed intriguing details. Note that a 36x40 inch plot is much more useful—a dense graph is easier to view on a larger printout. Dave Presotto described this smaller version as a smashed peacock on a windshield.

The map is colored using IP address; the first three octets of the IP number are used as the red, green, and blue color values respectively. This simplistic coloring actually shows communities and ISPs quite well.

We can already see features on this map: The fans at the edges show some interesting communities: Finland, AOL, some DISA.MIL, and Telstra (Australia and New Zealand). Looking at the color version of the map reveals a middle that is very muddled, showing our ISPs at the time: UUnet (green) and BBN (deep blue). SprintNet (sky blue) peeks through the sides.

The eye is drawn to a rather large star at the top of the picture, which represents the Cable and Wireless (cw.net) backbone, formerly the MCI backbone, formerly NSFnet. It is a major feature (if not the major feature) on the map. There are two reasons for this: (i) they are a huge backbone provider, and (ii) their backbone is an ATM network, connecting well over a hundred nodes around the world. Since our scanning is run at the IP level (level 3), this large network collapses to a single point. The smaller "Koosh" balls may be other ATM networks—we have not investigated this.

This map has changed over time, as we change our routing and ISP configurations. As we have done so, the predominant colors have changed as well.

We started collecting and preserving DNS names for the routers in March 1999. The collection of canonical names provides a rich source of data we can use to color the graph drawing. For example, colors can be selected based on top-level domain, showing the approximate country location of the hosts, or second-level domain, showing ownership of hosts.

## 4.1 Spanning tree plots

Though poster-sized versions of this map were quite beautiful (and quite popular), they did not really meet our original visualization goals. The middle was a mess, and it did not look like we could iterate our way out of it, so the resulting map was not particularly useful.

When we computed and plotted a minimum distance spanning tree (which we will define as a spanning tree of the original graph such that the distance from the root is preserved), the picture became much



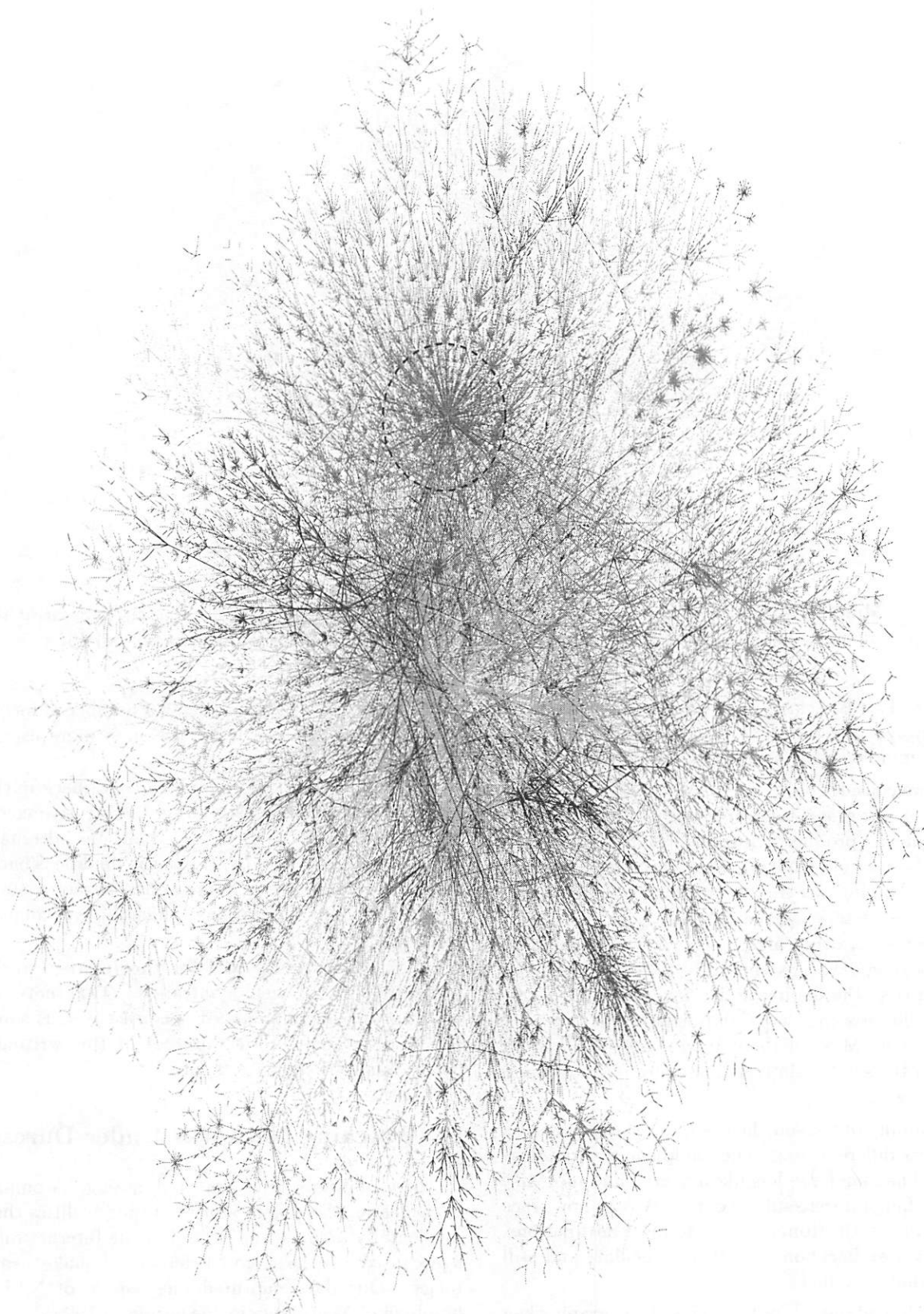


Figure 1: “Peacock-on-the-windshield” map from data taken in September 1998. The circled network is `cw.net`. Color versions of all figures are available at <http://www.cheswick.com/ches/map/mapfigs/>



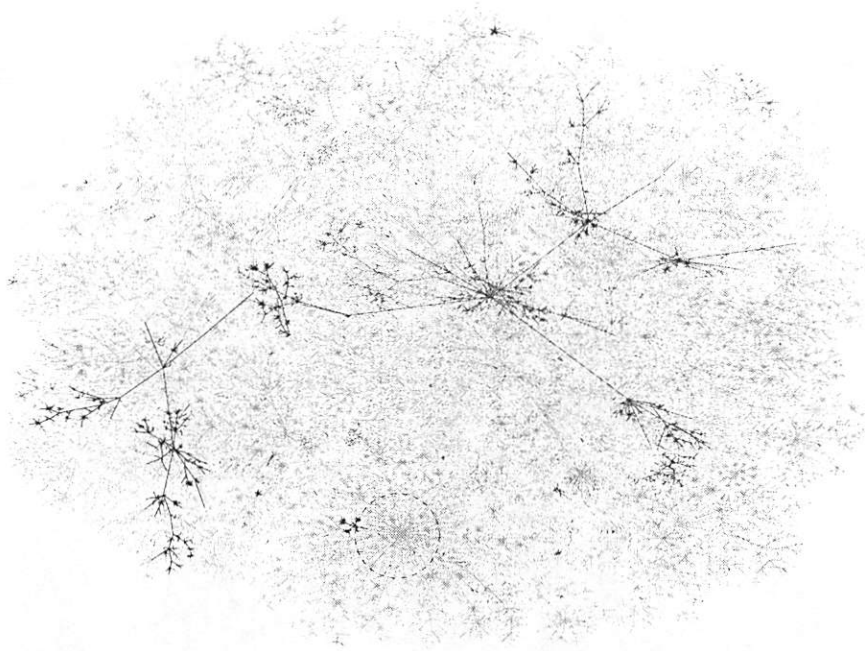


Figure 2: Minimum distance spanning tree for data collected on 2 November 1999. The circled star at the bottom is **cw.net**. The black foreground lines are links through net 12/8, Worldnet, one of our ISPs.

clearer. This is a cheat in one sense: our packets do not always take the shortest path. But the clutter in the middle cleaned up nicely (see figure 2).

If we consider only one shortest path to each destination, our graph turns into a tree, and the layout program can produce a much neater drawing. Alternatively, we could have used a graphing algorithm designed to lay out trees of arbitrary size [7], which tend to be faster than the general algorithms. Many of the tree drawing algorithms, however, result in unappealing drawings, due to two features of the resulting tree: the shallow nodes have very high degree (over 100, in some cases) and deep nodes have very low degree. Most of the standard algorithms work well for trees with relatively low degree trees (around two to ten).

Running our layout heuristic on the tree results in a very different map. The muddle in the middle is gone. The map looks less like a neuron and more like a coral fan or a space-filling curve. We can now trace individual paths from our host to most destinations. The **cw.net** backbone is still spectacular, and still somewhat muddled.

We lose about 5% of the edges of the graph when we throw away this inconvenient data. The edges still show interesting communities, but we can see

much more now. By eliminating a number of inconvenient edges, we can make the map more useful, and traceable by the eye.

Now we can add those missing edges back in the background, drawing them in an unobtrusive color, such as light cyan. In some cases, the alternate routes show up nicely. In others, the muddle is back, but out of harm's way. Some nodes attract a number of redundant connections, which the eye can pick out easily.

What works fairly well for the Internet works wonderfully for Lucent's intranet. That network has "only" 3,000 announced networks (versus some 90,000 registered for the Internet at this writing.) The full map is shown in figure 3.

## 5 Watching Networks Under Duress

Internet monitors have detected major disconnections before; there were stories of ping utilities that incidentally mapped the extend of the Internet outage caused by the Loma Prieta earthquake using pings. Our data captured one aspect of NATO bombing of Yugoslavia in the spring of 1999.

During the first month of the war few if any Internet links were cut. But in early May, the bombing

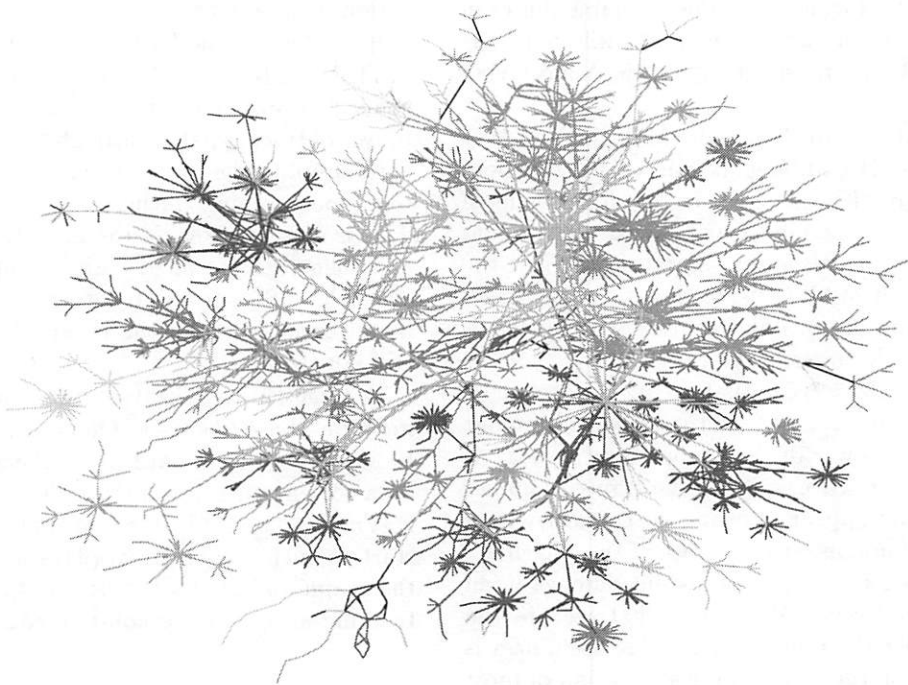


Figure 3: Lucent's intranet as of 1 October 1999.

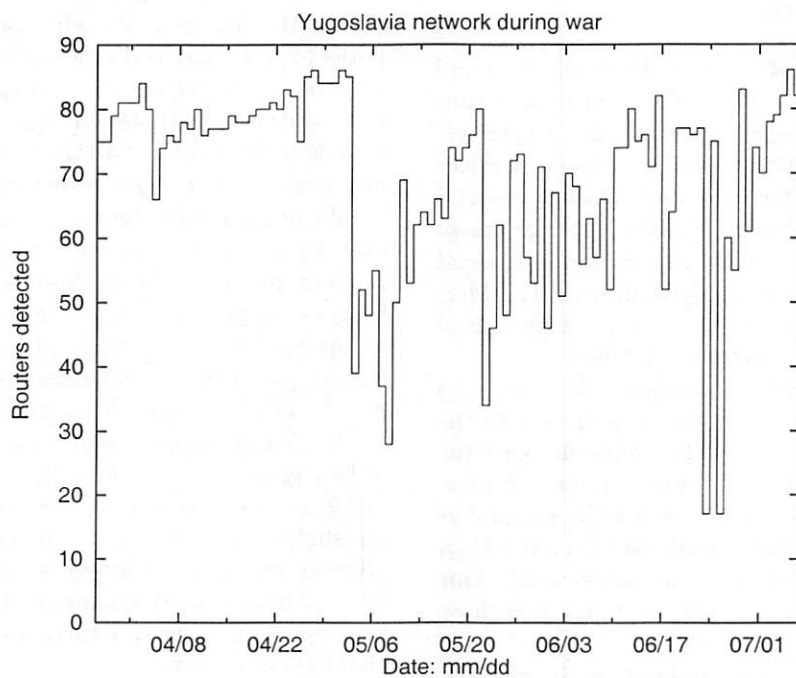


Figure 4: The number of reachable routers in the .yu domain over the course of the armed conflict.

moved to the power grid, and the resulting disconnection is clearly shown in figure 4. The connectivity returned slowly. Incidentally, the reachable routes in neighboring Bosnia also declined. We inferred (correctly) that Bosnia relies largely on the Yugoslavian power grid.

Figure 5 and figure 6 compare May 2nd and May 3rd of the NATO bombing. It is interesting to note two large spiny “Koosh” balls in the upper right of the map have been significantly reduced. This would seem to imply that although the core routers at the center of the “Koosh” balls were not directly damaged, many of the outlying routers were affected, possibly through power loss.

The maps also reveal that there appear to be only a few distinct routes into the Balkans from our test host. The power of the mapping technology is quickly apparent when viewing the limited number of gateways that appear, showing the connectivity of Yugoslavian domains with the rest of the Internet.

We detected the results of distant damage in an semi-automated way. We doubt that we are the first to consider the military uses. The usefulness is limited, because the exact physical location of most routers isn’t known. Related techniques will doubtless be useful for monitoring the extent of other natural disasters, particularly in well-connected parts of the world.

## 6 Related Work

There are a number of Internet data collection and mapping projects underway. Some have been running for a number of years, such as John Quarterman’s Matrix Information and Directory Services [18], which includes the “Internet weather report.” Martin Dodge has collected many representations of networks at Cybergeography [21]. Pansiot and Grad [12] mapped paths to 5,000 destinations. The Mercator project at USC [10] tries to get a picture of the Internet at a given instance in time.

In terms of long-term mapping, k claffy and CAIDA are collecting a number of metrics from the Internet with *skitter* [19]. They have mapped the MBone, and collected path data to major web sites. We choose to map to each known network, preferring to map to everything that exists, rather than everything that is used (i.e. the web servers). Our goal is to discover every possible path, not just those in use.

Internet maps are often laid out on the globe or other physical map. The desire to map the Internet to geography is compelling, but it tends to end up with dense blobs of ink on North America, Eu-

rope, and other well-connected regions<sup>2</sup>. However, connections to distant and more sparsely connected regions can be represented nicely, *c.f.* Quarterman’s map of connections to South America.

The problem with this method is the well-connected areas remain thoroughly inked, without a prayer of tracing paths through them. One approach is to simplify the map, showing connections by autonomous systems rather than individual routers. This is akin to showing the interstates on one map, and then creating local maps for each state. However, the AS connectivity graph is, proportionally, more connected than the IP graph, so the graph is still not very legible.

Interactive visualization tools can aid in navigating a database like ours. One can zoom, query, and browse at will. It is hard to see the entire net clearly on a screen: there are far too few pixels. However, H3Viewer [11] [17] is one tool that looks like a good start to such a tool. It displays a spanning tree of the graph and allows the user both to navigate the tree and also view the non-tree edges.

## 7 Future Work

At present, we are scanning out from a single test host. If we run the same scans from multiple hosts throughout the world, we will discover many more edges, and create a more accurate map of the “middle” of the Internet. We will discover the incoming paths to test hosts from the outgoing paths of other test hosts. Clearly, we need to expand the number of test locations. If we use enough of these, we should be able to fill in almost all the links that we can’t see now because we never use them in out-going paths.

We originally thought that we would need to locate computers world-wide, or obtain volunteers to run our mapping. Jorg Nonnenmacher suggested that we might offer a screen saver that displays an updated network map, and would perform modest mapping chores from sites scattered all over the world when instructed from a central site.

Jorg’s suggestion is seductive, but it would have to be engineered very carefully to avoid abuse. The real problem, however, is that the tracing packets are slightly noxious. It would be best if we could preserve the return address, so they always appear to come from `ches-netmapper`. This makes filtering and reporting easier for those who watch and care about these packets.

<sup>2</sup>Producing a map distorted based on Internet connectivity in order to alleviate this problem might be an interesting problem for some cartographer.

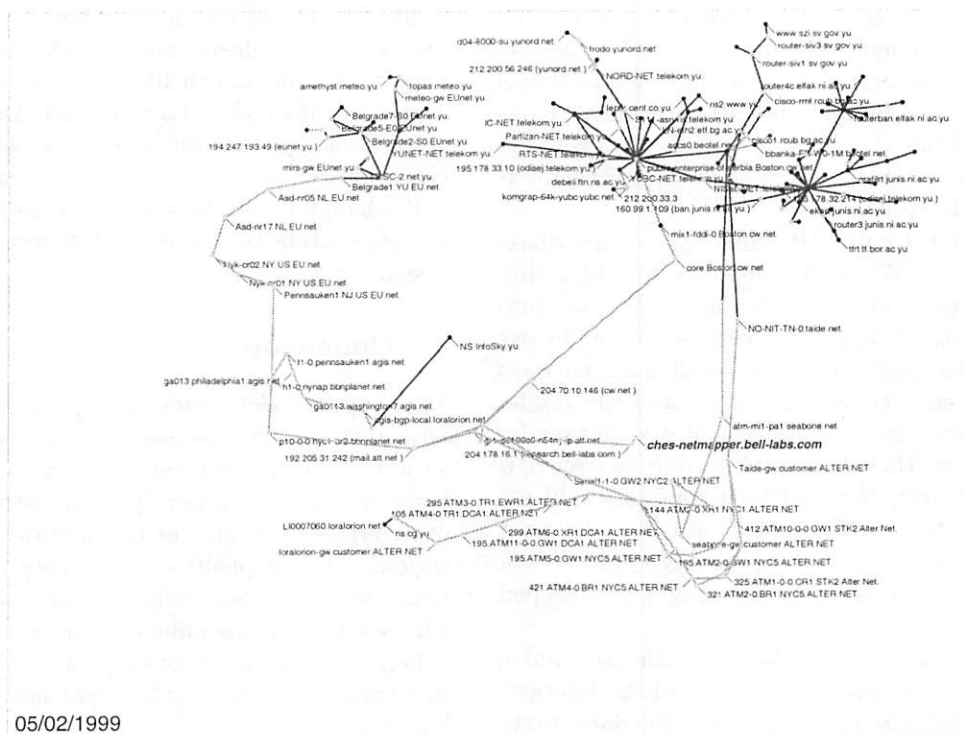


Figure 5: Map of paths to the Yugoslavian networks on May 2, colored by network.

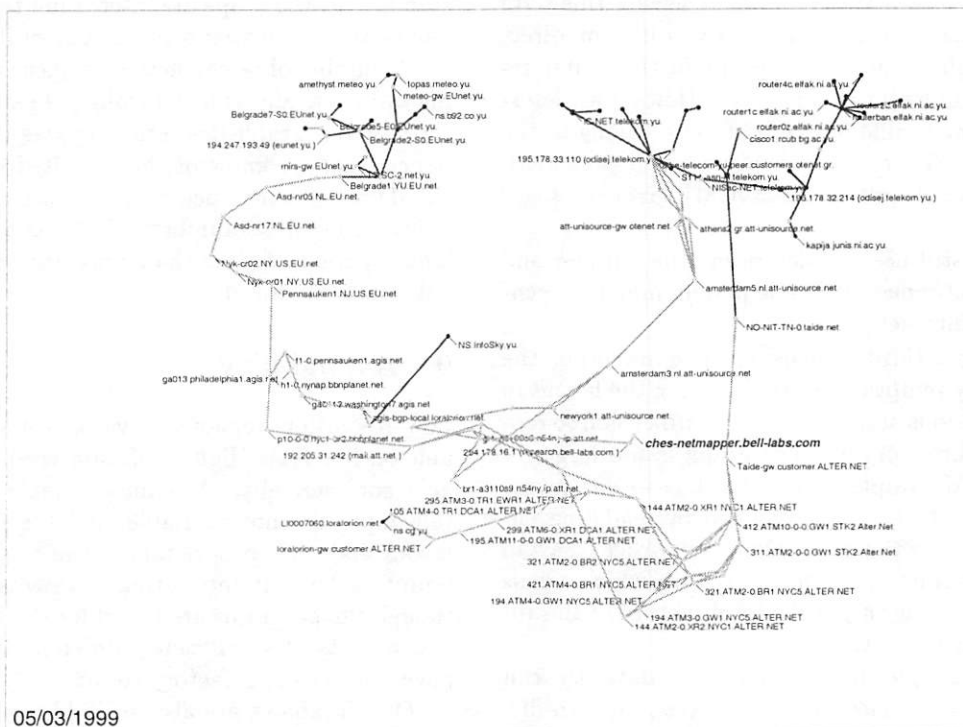


Figure 6: Map of the Yugoslavian Internet on May 3, colored by network. The main hubs in the upper right are still reachable, but they have lost a lot of leaves.



Others have suggested that we use loose source routing to guide the probe packets down the desired paths. Though some have reported some success with this approach [10], we have found that a large majority of the Internet either blocks IP packets with options, or at least refuses to process them. We could display these nodes on our map—an interesting visualization.

We intend to use IP tunneling to distribute probe packets. We need volunteers to add a simple tunnel to their router for us. Then we tunnel packets to their router, with return addresses of `ches-netmapper`. Packets would trace outward paths from each tunneling router, and the results neatly returned to us. Sensitive sites would see familiar packets, though they may come in over new links. Of course, the tunneling routers would see each packet twice. These wide scans would need a lot more packets, so we probably couldn't run them daily. Also, such packets might be dropped by ingress filters.

The resulting data ought to enable us build a mesh that closely describes the core of the Internet. We are not yet sure how to plot it—the data surely will look like our “peacock” and will need reduction or interactive visualization tools. And our layout tool only works on rooted trees at the moment.

There is also a tricky problem sewing this data together. Traceroutes going in two different directions through a router may result in the router reporting two different IP addresses. How do we determine that those different IP addresses belong to the same router? There are several possibilities, including looking at the return IP of ICMP error messages [10].

We will still need to determine the number and position of sites needed to adequately map the “center” of the Internet.

Utilizing a third dimension in representing the graph is very tempting, either by doing the layout in three dimensions or using the third dimension to represent distance from us. The graph is too large for current VRML implementations that we are aware of, but ought to be easily handled by rendering engines. The other major problem is in order to avoid “background clutter,” fog must be used, which means that a viewer can only see a local picture of the Internet at any given time.

Several people have taken our data to run through their visualization tools. Alas, modern displays simply lack the pixels to display the whole thing at once without some form of abbreviation. We look forward to their results.

We now have almost two years of data concern-

ing the Internet. We would like to create a movie of how the Internet's topology has changed over our dataset. The problem is making the picture for January 12th look enough like the picture for January 11th that the movie is fairly smooth while still showing a decent picture for both days. This is complicated by the fact that companies change ISPs and ISPs change internal connectivity, peering arrangements, routing decisions, and router - IP address assignments.

## 8 Conclusion

The mapping technology can reveal insights about large networks. We've used these tools on intranets as well, to help show our company's connectivity. Some intranet maps clearly show routing leaks and other errors. We have used colors to show insecure regions, new acquisitions, and rare domains (domains with very few mapped hosts), which usually denote a leak or misconfiguration. The maps helped debug our corporate routing table, which contained route announcements for `lsu.edu` and the US Postal Service.

The Internet maps, while seemingly less useful have certainly excited the media, who lacks good visuals of the Internet [25] [26]. From a less scientific standpoint, the maps are interesting to look at, and one publisher created a poster out of it [27].

A number of researchers have picked up the routing database and run it through their visualization tools or run graph-theoretic analyses of it, and one paper (that we know of) has resulted so far [3]. As the data collection began in August, 1998, it provides a good deal of information about routing for a longer period of time than most routing studies to date have employed.

## 9 Availability

Low resolution versions of various maps are available on-line [22]. High resolution versions are available commercially. Machine-readable high resolution maps are not available, and the mapping and layout code are proprietary. The authors will attempt to lay out interesting data sets on request, though the programs are tuned for the Internet data and layouts of significantly different types of data have not been satisfactory so far.

Our databases are also available at our web site, both the label database and the route database. Historic and current databases are available, along with the explanation of the database format from appendix A.

## 10 Acknowledgments

Tamara Munzner, Stephen North, and Steve Eick have guided us into the world of visualization algorithms and tools. k claffy, Daniel McRobb, and the rest of the folks at CAIDA have helped us with mapping issues and ideas. Tom Limoncelli suggested the simple web server, and helped with Lucent and Internet routing issues.

Tom Limoncelli, Bob Flandrina, Paul Glick, and Dave Presotto all have their names connected to the network that houses our test host, and have had to field queries and complaints about this project. We thank them for their continuing good humor to do so.

## References

- [1] Burch, H. and Cheswick, W., "Mapping the Internet," IEEE Computer, Vol. 32, No. 4, April 1999.
- [2] Burch, H. and Cheswick, W., "Tracing Anonymous Packets to Their Source by Selective Denial-of-Service Probes," *submitted to LISA*.
- [3] Cheswick, W., Nonnenmacher, J., Sinha, R., and Varadhan, K., "Modeling Internet Topology," Submitted to ACM Sigmetrics 2000.
- [4] Claffy, K., *et al*, "Internet Tomography," Nature, January 7, 1999.
- [5] claffy, k., private communication.
- [6] Cohen, J., "Drawing graphs to convey proximity: an incremental arrangement method," ACM Transactions on Human-Computer Interaction 1997, v.4, no.3, p.197-229.
- [7] Di Battista, G., *et al*, "Graph Drawing," p41-64, Prentice-Hall, 1999.
- [8] Eades, P., "A heuristic for graph drawing," Congressus Numerantium, Vol. 42 p.149-160, 1984.
- [9] Frick, A., Ludwig, A., and Mehldau, H., "A fast adaptive layout algorithm for undirected graphs," Proceedings of Graph Drawing '94, 1995.
- [10] Govindan, R. and Tangmunarunkit, H., "Heuristics for Internet Map Discovery," Technical Report 99-717, Computer Science Department, University of Southern California.
- [11] Munzner, T., "H3: Laying Out Large Directed Graphs in 3D Hyperbolic Space," Proceedings of the 1997 IEEE Symposium on Information Visualization, October 20-21 1997, pp 2-10, 1997.
- [12] Pansiot, J.-J. and Grad, D., "On routes and multicast trees in the Internet," Proceedings of IEEE INFOCOM '97, Apr 1997.
- [13] Russel, S. and Norvig, P., "Artificial Intelligence: A Modern Approach," p. 111-114, Prentice-Hall.
- [14] Shewchuk, J., "An Introduction to the Conjugate Gradient Method without the Agonizing Pain," Carnegie Mellon University, School of Computer Science, unpublished paper
- [15] Stevens, W. Richard, *TCP/IP Illustrated Volume 1*, Addison Wesley, 1994, pps. 97-110.
- [16] Tunkelange, D., "JIGGLE: Java Interactive Graph Layout Environment," Proceedings of Graph Drawing '98, 1998.
- [17] <http://graphics.stanford.edu/munzner/h3/>
- [18] <http://www.mids.org/>
- [19] <http://www.caida.org/>
- [20] <http://www.internetweather.com/>
- [21] <http://www.cybergeography.org/>
- [22] <http://www.cs.bell-labs.com/~ches/map/index.html>
- [23] <http://www.cs.bell-labs.com/~ches/map/db.gz>
- [24] <http://www.cs.bell-labs.com/~ches/map/lucent.mpeg>
- [25] *Wired*, December 1998.
- [26] "Cartography", *National Geographic*, Jan. 2000.
- [27] <http://www.peacockmaps.com/>
- [28] <ftp://ftp.merit.edu/routing.arbiter/radb/dbase/>

## A Database format and details

Each day's run produces three files: the path database, an updated list of router names, and a log. Each is in text form, suitable for processing by traditional UNIX filters. All three files are archived for long-term reference.

The log contains the collection information, with some lines containing a Greenwich time stamp.

### A.1 Path database

The path database contains one line per target network, and is divided into fields separated by white space. The first field is the target network, in a familiar form:

```
135.104.0.0/16
```

The filters assume that all four octets are present.

The remaining fields are in the form:

```
<name>=[<date>:]value
```

where <date> has the form `yyyymmdd`, suitable for sorting (although not Y10K compliant).

The field types are listed below. Only the first four appear in current databases—the rest are deprecated and have not been used since fall 1998. Some fields may appear more than once, representing data collected at different times. They are usually sorted by date.

Name	Date	Value	Description
Path	yes	see below	path to target
Probe	yes	(none)	date of last test
Target	yes	IP addr	host on target net
Whiner	yes	email addr	don't scan this net
Asnpath	no	unused	deprecated
Name	no	net owner	not used
Complete	no	(none)	deprecated
Pathdate	no	date	deprecated

The path field contains a comma-separated list of IP numbers, possibly followed by a completion code. If no code is present, the path reached the target. The other completion codes are:

?	same as !?	deprecated
!F	ICMP filtered	firewall encountered
!H	ICMP host unreach.	bad guess for the target
!N	ICMP net. unreach.	firewall, filtered, etc
!R		routing loop, deprecated
!L		routing loop
!Z	incomplete	deprecated
!!	incomplete	deprecated
!?	incomplete	no response

### A.2 Label database

The label database has one entry per line. Each entry has three fields separated by white space: an IP number, a label, and the date (as `yyyymmdd`) it was collected.

The label consists of a name as returned by a DNS PTR lookup. If a domain nameserver reported "no such domain," the domain of that nameserver is given in parenthesis. This gives some idea of who owns the IP address. If there is no answer, the label is the IP number enclosed in less-than/greater-than symbols: `<135.104.53.2>`.



# Measuring and Characterizing System Behavior Using Kernel-Level Event Logging

Karim Yaghmour and Michel R. Dagenais  
*Département de génie électrique et de génie informatique*  
*Ecole Polytechnique de Montréal*  
*C.P. 6079, Succ. Centre-Ville*  
*Montréal, Québec, CANADA, H3C 3A7*  
karym@opersys.com, michel.dagenais@polymtl.ca

## Abstract

Analyzing the dynamic behavior and performance of complex software systems is difficult. Currently available systems either analyze each process in isolation, only provide system level cumulative statistics, or provide a fixed and limited number of process group related statistics. The Linux Trace Toolkit (LTT) introduced here provides a novel, modular, and extensible way of recording and analyzing complete system behavior. Because all significant system events are recorded, it is possible to analyze any desired subset of the running processes, and for instance distinguish between the time spent waiting for some relevant event (data from disk or another process) versus time spent waiting for some unrelated process to use up its time slice.

Despite the extensive information gathered, experimental results show that the LTT time and memory overhead is minimal ( $< 2.5\%$  when observing core kernel events). Moreover, due to the LTT and Linux kernel modularity and open source code availability, the system is easily extended both in terms of system events gathered, and of later post-processing and graphical presentation.

## 1 Introduction

System performance measurement, for understanding and optimization, may be performed at different levels according to varying needs. For example, system administrators need to understand what is using up all the resources and where the bottlenecks are; this may help determine which hardware upgrade would be most beneficial, or who is abusing the system. On the other hand, the application user or developer wants to understand where all the

elapsed time is spent; this may include several related processes as well as system services such as network and file systems.

At the system level, easily available performance data usually includes the average load, number of interrupts, number of packets sent and received, and number of disk blocks read and written. At user level, it is possible to obtain per process and per process group the elapsed time, system and user CPU time, number of input/output operations, instruction counts and portion of CPU time spent in each function.

This information is sufficient to analyze the overall system performance or single CPU intensive processes. However, a different approach is required to analyze complex networked multi-process software systems, running on multi-processor systems, and taking into account the aggressive disk caching policy of modern operating systems. An excellent example of such a complex performance problem is the recent discussions on the Linux kernel mailing lists about the lower than expected performance on very high-end web servers. After tedious ad hoc manual code instrumentation and repeated experiments, they eventually focused on two perceived problems: coarse grain locking in the TCP/IP stack and the thundering herd problem (several daemons were awakened when new data arrived while a single daemon could use the data anyway). Such complex and specialized applications motivate the need for a comprehensive yet low-overhead, modular, and extensible event monitoring system for detailed performance analysis.

In section 2, existing profiling and measurement tools are reviewed and discussed. Section 3 details the architecture of the new Linux Trace Toolkit

(LTT) presented here. Section 4 presents the overhead caused by LTT. Section 5 presents cases where LTT has been used to analyze system behavior and compares LTT to existing analysis tools. Section 6 discusses possible future directions.

## 2 Related work

As discussed in the previous section, there are two broad categories of existing profiling tools. The first is aimed at the detailed analysis of individual applications and the second is aimed at an overview of the system's behavior.

In the first category, we find tools such as DCPI [9], Morph [11], Path Profiler [6], Quantify [1] and GProf [12], to name a few. DCPI provides a detailed analysis of different processes running on a system down to pipeline stalls. In order to provide its highly detailed data, DCPI uses a very high frequency interrupt. Similarly, Morph uses the clock interrupt to gather data in order to optimize applications off-line. Both systems fail to provide their user with information on the interactions of the different processes. Neither enable the user to understand the dynamics of the observed system. Path Profiler is based on work done by J. Larus and T. Ball [8] and, contrary to DCPI and Morph, is an instrumentational approach to data sampling. Path Profiler is much like GProf but is much richer in detail. Here, the problem is the overhead. By the authors' own evaluation, the overhead is around 30%. On a normal running system, this is often not tolerable. As reported in [6], Quantify uses techniques similar to Path Profiler to provide profiling information, but its capabilities remain confined to analyzing one process at a time. Moreover, its overhead is unpublished.

All these profiling systems provide detailed analysis of one or many processes, but fail to provide information on system dynamics. Yet, two of these systems bare a significant contribution that is used by LTT and that provides an efficient way to collect a high volume of data without hindering system performance. DCPI and Morph use a combination of kernel modifications, driver, and daemon to collect the necessary data for their operation. This is a departure from the traditional practice of using new system calls or adding entries to the */proc* directory and provides a wealth of opportunities for profiling and measuring systems.

On the other end of the spectrum, we find such tools as UNIX *ps* and *top* and the Windows NT Performance Monitor [15]. The former often use the content of the */proc* directory to present the user with system statistics. The later uses system calls that enable the user to read into kernel counters. Both are based on sampling or on crude event counting. Both lack the possibility to track the order in which events occurred or their details. They cannot, therefore, be relied on to offer a correct appreciation of the underlying system's dynamics.

In a category of their own, we find specialized tools that enable the user to track key system events while preserving their order of occurrence and some details. These tools have primarily been used for debugging or for security auditing. They have not been designed for measurement or characterization. For instance, WindView [2], DejaView [3] and the Hyperkernel Trace Utility [4] are primarily designed to help embedded system designers understand the dynamics of their systems in order to clarify synchronization and resource usage problems. The security auditing tools are limited by their own purpose. That is, their usage cannot be generalized to purposes other than security auditing. Neither category is intended to provide the user with system performance analysis. Moreover, the debugging tools, which are far more elaborate than the security oriented tools mentioned, are all proprietary. Therefore, they cannot be improved and extended in the same way by the user community.

Another approach is used in SimOS [13] which simulates the hardware on which an operating system runs in order to retrieve information regarding its behavior and how applications interact. This, though, remains a simulated system and cannot be used on production systems. In this regard, LTT is not unique in the information it provides. Yet, it is the only tool available for a mainstream operating system that has been designed from the ground up in order to provide the user with not only a view of the dynamic behavior of the system, but also a characterization of its behavior and a measurement of the different latencies suffered by and because of the different processes running. Moreover, it is an open-source project and can, therefore, be modified and extended under the GNU General Public License [5].

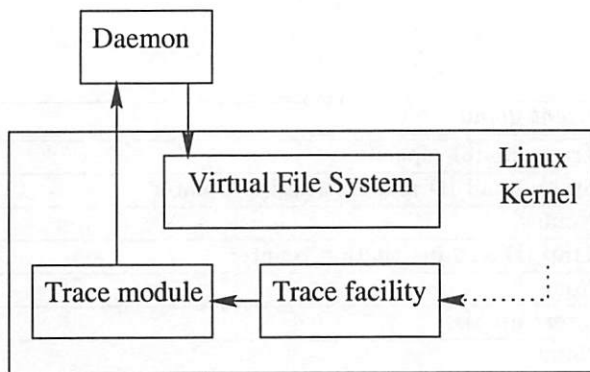


Figure 1: LTT architecture.

### 3 Toolkit architecture

LTT is composed of independent software modules. Each module has been designed in order to facilitate extension while minimizing performance overhead. Figure 1 represents the architecture used. Note that, for reasons of simplicity, only the architectural parts of the Linux kernel relating to LTT's functionality are presented. Details for the Linux architecture can be found in [10, 14]. Details for a general UNIX system architecture can be found in [16, 7]. The arrows indicate the flow of information through the different modules making up LTT. The primary source of information being the instrumented kernel. The source of this primary information is not attributed to a kernel component in particular since many components convey trace information. Therefore, not all the modules producing information are illustrated, to simplify presentation, but they are all discussed in detail below.

Basically, events are forwarded to the trace module via the kernel trace facility. The trace module, visible in user space as an entry in the `/dev` directory, then logs the events in its buffer. Finally, the trace daemon reads from the trace module device and commits the recorded events into a user-provided file.

Section 3.1 discusses the kernel trace facility. Section 3.2 discusses the instrumentation of the kernel. Section 3.3 discusses the trace module. Section 3.4 discusses the trace daemon. Finally, section 3.5 discusses the data analysis and presentation software that come with LTT.

#### 3.1 Kernel trace facility

The kernel trace facility is an extension to the core kernel facilities. Its function is to provide a unique entry point to all the other kernel facilities that would like an event to be traced. However, it does not log the events. Rather, it forwards the trace request to the trace module.

To achieve such functionality, the trace module has to register itself with the trace facility upon system startup, if it was compiled as part of the kernel. When compiled and loaded as a separate module, the registration will take place when the module is loaded. When registering, the trace module provides the trace facility with a call-back function that is to be called whenever an event occurs. If no trace module is registered, then the traced events are ignored. Furthermore, it provides the trace module with the possibility to configure the way the instruction pointer values are recorded upon the occurrence of a system call. For instance, since most system calls are done from a loaded library rather than from the code belonging to the running application, it is a desirable feature to specify either the number of call depths on the stack or an address range from which the instruction pointer should come from. Once set, the kernel will browse the stack to find an instruction pointer matching the desired constraints, whenever a system call occurs.

In summary, the kernel trace facility acts as a link between the trace module and the different kernel facilities.

#### 3.2 Kernel instrumentation

The kernel instrumentation consists of the different events being traced in the kernel. There are different types of events. Each type of event has its own data set and, as such, the length of a trace entry varies according to the type of event recorded. Within a given type, there can be different subtypes. This allows various degrees of detail. Figure 2 lists for every type of event existing subtypes and the recorded data.

Given the number of modifications to the kernel source code and the number of files modified, a set of macros has been used instead of a direct call to the services of the trace facility. During compilation of the kernel, if the tracing is disabled in the configuration, the added code will have no effect. If



<i>Event type</i>	<i>Event subtype</i>	<i>Event detail</i>
Trace start	None	Trace module specific
System call entry	None	System call ID and instruction pointer
System call exit	None	None
Trap entry	None	Trap ID and instruction pointer
Trap exit	None	None
Interrupt entry	None	Interrupt ID
Interrupt exit	None	None
Scheduling change	None	Incoming task, outgoing task and outgoing task's state
Kernel timer	None	None
Bottom half	None	Bottom half ID
Process	Create kernel thread	Thread start address and PID
	Fork	PID of created process
	Exit	None
	Wait	PID waited on
	Signal	Signal ID and destination PID
	Wakeup	Process PID and state before wakeup
File system	Buffer wait start	None
	Buffer wait end	None
	Exec	File name
	Open	File name and descriptor
	Close	File descriptor
	Read	File descriptor and quantity read
	Write	File descriptor and quantity written
	Seek	File descriptor and offset
	Ioctl	File descriptor and command
	Select	File descriptor and timeout
	Poll	File descriptor and timeout
Timer	Expired	None
	Set itimer	Type and time
	Set timeout	Time
Memory	Page allocate	Size order
	Page free	Size order
	Swap in	Page address
	Swap out	Page address
	Page wait start	None
	Page wait end	None
Socket communication	Socket call	Call ID and socket ID
	Socket create	Socket type and ID of created socket
	Socket send	Type of socket and quantity sent
	Socket receive	Type of socket and quantity received
Inter-process communication	System V IPC call	Call ID and entity ID
	Message queue create	Message queue ID and creation flags
	Semaphore create	Semaphore ID and creation flags
	Shared memory create	Shared memory ID and creation flags
Network	Incoming packet	Protocol type
	Outgoing packet	Protocol type

Figure 2: Kernel events traced.

tracing is selected, then the macros will be replaced by the portion of code that sets the desired values and calls the trace facility.

### 3.3 Trace module

The trace module is a key element of the architecture. The performance of the trace process largely depends on it. In theory, the goal of the trace module is simple: store the incoming event descriptions and deliver them efficiently to the trace daemon. In practice, its implementation is much more elaborate. There are many reasons for this.

First, the trace module must retrieve additional information for each event. This additional information consists of the time at which the event occurred and the CPU identifier. Since the absolute time is large (an 8 byte pair consisting of seconds and microseconds), only the time difference between the current event and the time at which the last buffer switch occurred is recorded. The time of the last buffer switch is necessary to support a double buffering scheme explained below. Given the size of the buffers and the frequency of event occurrences, 4 bytes suffice. Note that the trace module uses the *do\_gettimeofday* kernel call in Linux in order to obtain the absolute time at which an event occurred. Under Pentium type PCs, this call uses the processor's Time Stamp Counter (TSC) which enables microsecond precision on timestamps.

Second, the trace module must be configurable. The following configuration options are possible:

- Set event data buffer size.
- Set event mask.
- Set event details mask.
- Record CPU ID.
- Track a given PID.
- Track a given process group.
- Track a given GID.
- Track a given UID.
- Set call depth at which the instruction pointer is to be fetched for a system call.
- Set an address range from which the instruction pointer is to be fetched for a system call.

The event mask is used to determine whether an event is to be logged. The event details mask is used to determine whether the details of a given event are to be recorded. Tracking a PID, process group, GID, or UID, will result in the logging of only the events occurring during the execution of a process fitting the right description. The call depth and address range for a system call have previously been discussed. Configuration is done through the *ioctl* system call.

Third, the trace module must be reentrant since an event can occur from two different levels of priority at the same time. For instance, the trace module could be dealing with a system call event when an interrupt occurs. Given the fact that interrupts have priority over any other event and that different interrupts have different priorities, the processing of the system call will be suspended and an interrupt event will call upon the trace module. To solve this issue a kernel lock is used during the critical part of the event treatment. This though raises another concern. Holding a kernel lock for an extended period of time can be costly given the frequency at which some events can occur. This is not the case with LTT, since the kernel lock used is held only for the time of the logging of the event into the module's buffer. Furthermore, since the logging procedure does not call upon any other procedure, there can be no priority-inversion problem. Each event logging is interruptible once it has released the kernel lock by any higher priority event.

In order to be accessible through the virtual file-system as a device, the trace module provides a basic set of file operations. These are:

- *open*, upon which a pointer to the task structure of the caller is kept in order to be used by the trace module when the trace buffer is full.
- *ioctl*, to provide the daemon with a way to configure tracing.
- *release*, for the device to be released when the daemon dies or when the device is closed.
- *fsync*, to reset the driver.

To efficiently deal with the large quantity of data generated, the trace module uses double-buffering. That is, a write buffer is used to log events until it reaches its limit. At which time, the daemon is notified using a SIGIO signal. Once the write

buffer has been filled, the trace module assigns it as the read buffer and uses the previous read buffer as the new write buffer. Of course data can be lost if the daemon is not rescheduled before the new write buffer fills. Since the size of the buffers used by the module is configurable, it is up to the daemon to configure the module properly. Moreover, even if the daemon has opened the trace module, events do not get logged until the daemon uses the start command via *ioctl*. Logging may be discontinued with the stop command.

### 3.4 Daemon

The primary function of the daemon is to retrieve the information accumulated by the trace module and store it, typically in a file. It provides the user with a number of options to control the tracing process. In addition to giving the user access to the options available from the trace module, the daemon lets the user specify the tracing duration.

Once the daemon is launched, it opens and configures the trace module, and sets a timer if a time duration was specified. Otherwise, the user must kill the daemon process manually to stop the trace. In normal operation, the daemon sleeps awaiting SIGIO signals to read from the trace module, or timer/kill events to end tracing.

Akin to the trace module, the daemon uses double-buffering. Here, though, the intent is not on preventing the loss of events but on reducing the impact on the system, due to frequent reading and writing, and reducing the pollution of the trace, due to the daemon using system resources. Therefore, when it receives a SIGIO signal from the trace module, the trace daemon reads the content of the module's buffer and appends it to the content of an internal buffer. Once this buffer is full, it is committed to file and, while doing so, a second buffer is used to record the incoming data. This second buffer will be used as the first one was. When tracing is finished, the trace module and the trace data file are closed. Note that even though the daemon writes to a file, this does not necessarily mean that information gets written to disk. In fact, given the different caching mechanisms used by Linux, the information is written to disk somewhat later in big chunks by the *kupdate* kernel thread.

Although the data collection method described above provides the detailed dynamics of the system,

there is one more piece of information missing, the system's state prior to the trace start. To this end, the trace daemon will go through the */proc* directory recording for each process the following characteristics: process ID (PID), name (as given to *exec*) and parent's PID (PPID). This is done following the configuration of the trace module and the start of the trace. The information retrieved is stored in a file that will later be used by the analysis software.

### 3.5 Data analysis and presentation software

Unlike the other LTT components, the data analysis and presentation software is typically run off-line. It uses both the initial processes state and the trace data files created by the daemon to recreate the dynamic behavior of the system in the observed time interval.

The initial state file is used to create the process tree as it was before the trace started. The trace file is then used as the trace event database. This is accomplished using the *mmap* system call and a collection of primitives that enable the extraction of information regarding the events from the trace. These functions provide the following services: get an event's ID, its time of occurrence, the process to which it belongs and the human-readable string describing it. Also, they enable forward or backward browsing of the events trace. An important functionality they provide is to determine whether an event is a control event or not. By control event, we mean an event that results in the transition of control from or to the kernel. A system call, for instance, always marks a control transition to the kernel. A trap, on the other hand, might not mark a transition since traps can occur during the execution of kernel code. It is not the objective of this paper to present the complete conditions under which control transitions occur under Linux, but these conditions have been formalized in the course of the development of LTT in order to facilitate the reconstruction of the event graph and the computation of the different performance measures.

An important component of those primitives is the trace analysis procedure. This procedure reads the entire trace cumulating results about the behavior of the system during the trace. This is where the core of the trace processing takes place. Here, the time spent scheduled, the time spent executing process code, the time spent in system calls, the num-

ber of occurrences of the different events, etc. are computed.

These primitives serve as the basis for the analysis software which is usable either from the command line or from a GUI. The GUI front-end enables the user to browse the trace in both graphical form and list form. Both interfaces enable the presentation of the cumulated system statistics and the list of events. They also enable the user to select the events to present and the events to ignore. Moreover, the graphical front-end provides a search menu which simplifies event searching.

The possibility to view the trace in graphical form is interesting since it enables the user to easily view interactions that are often deemed complex. He can then follow the flow of control and clearly identify the different transitions.

## 4 Toolkit overhead

The main novelty of LTT, besides the extensibility provided by the modular open source architecture, is the extent of available information while remaining a non-invasive low overhead monitoring tool. Thus, this section concentrates on the study of the overhead caused by LTT.

To this end, section 4.1 deals with quantifying the overhead caused by LTT. Section 4.2 discusses the ramifications of the observed overhead. Section 4.3 presents LTT's memory footprint. Finally, section 4.4 discusses possible improvements.

The experimental results show that LTT is able to provide unique data sets with very little overhead on the observed systems. Typically, an observed system incurs less than 2.5% overhead when monitoring core system events.

The experiments were performed on an Intel Pentium II 350MHz processor with 128MB of main memory. The Linux distribution used is RedHat 6.0. Of course, the default kernel is replaced by the LTT modified Linux kernel. Unless stated otherwise, all jobs were run from a plain command-line outside any graphical environment. The daemon is configured to use 1,000,000 bytes buffers and the trace module is configured to use 50,000 bytes buffers. Therefore, the daemon will receive a signal from the trace module every time 50,000 bytes of trace are generated and it will commit data to file every

1,000,000 bytes of trace data. The times for these to occur vary according to the job traced and are presented below.

### 4.1 Overall system overhead

In order to evaluate the overhead caused by LTT to a running system, a number of modifications had to be made in order to isolate the impact caused by each component to the overall architecture. The following configurations were tested:

1. Original 2.2.13 kernel (base configuration).
2. Modified 2.2.13 kernel. Events are ignored by the kernel trace facility.
3. Modified 2.2.13 kernel. Events are logged by the trace module but the daemon is not running.
4. Modified 2.2.13 kernel. Events are logged and reported, but the daemon is not writing the data to a file.
5. Modified 2.2.13 kernel. Events are logged, reported and written.
6. Same as above, except that event details are only recorded for core kernel events <sup>1</sup>.

On each system, a batch of jobs was issued and the elapsed time to complete every job recorded. Thereafter, job completion times could be compared and provide us with insight on which component was slowing down the system, if any. To this end, the following jobs were issued using shell scripts:

1. Complete compilation of a Linux kernel, using make. This job is CPU and file intensive.
2. Creation of a tar archive using a large number of files (800MB). This job is file intensive.
3. Compression of an archive of the Linux kernel source (50MB), using bzip2. This job is CPU intensive.
4. While running the KDE environment, the following tasks were started twice in parallel:
  - (a) *netscape* loaded different web pages from 14 different sites.

<sup>1</sup>Incidentally, the core kernel events are the events presented in Figure 2 that have no subtypes.



Conf.	Compile	Archive	Compress	Desktop
1	240.2	357.4	141.2	246.8
2	240.8	358.0	141.4	249.2
$\Delta$	0.25 %	0.17 %	0.14 %	0.97 %
3	243.6	359.1	141.1	252.2
$\Delta$	1.42 %	0.48 %	-0.07 %	2.19 %
4	245.7	358.1	141.6	252.9
$\Delta$	2.29 %	0.20 %	0.28 %	2.47 %
5	246.9	365.5	141.4	258.3
$\Delta$	2.79 %	2.27 %	0.14 %	4.66 %
6	246.3	363.6	141.6	252.9
$\Delta$	2.54 %	1.74 %	0.28 %	2.47 %

Figure 3: Job completion times in seconds according to configuration.

- (b) *acoread* opened 7 different documents.
- (c) *staroffice* opened 8 different documents.
- (d) *gnuplot* drew 4 functions 14 times.

This job is I/O and operating system<sup>2</sup> intensive. It represents the extreme case of user desktop usage. Note that for this job, the daemon is configured to use 2,000,000 bytes buffers and the trace module is configured to use 250,000 bytes buffers.

The results of these tests are presented in Figure 3 and discussed in section 4.2. For each job-configuration pair, except the ones belonging to the first configuration, the percentage difference between its completion time and the one for the base configuration is given. This facilitates further analysis. The times given are in seconds and represent the average time over 10 runs<sup>3</sup>. Note that due to the complexity of modern UNIX systems the times reported vary. Figure 4 presents the standard deviation for each of the times reported. Figure 5 presents the size of the traces generated and the rate of generation of the traces. These results will be discussed in section 4.3.

<sup>2</sup>The OS has to deal with scheduling many competing processes besides having to manage inter-process communication to the X server via sockets.

<sup>3</sup>Each test was actually run 11 times, the results from the first run being discarded.

Conf.	Compile	Archive	Compress	Desktop
1	0.4	1.6	0.4	0.9
2	0.6	1.9	0.5	1.9
3	0.5	1.7	0.3	1.6
4	0.5	2.0	0.5	1.1
5	1.1	1.6	0.5	3.9
6	0.5	1.3	0.5	1.8

Figure 4: Standard deviations of the measured job completion times in seconds according to configuration.

Job	Conf.	One run (MB)	Rate (MB/s)	Read Freq. (Hz)	Write Freq. (Hz)
Compile	5	33.4	0.135	2.83	0.14
	6	25.0	0.101	2.12	0.11
Archive	5	27.9	0.076	1.59	0.08
	6	16.5	0.045	0.94	0.05
Compress	5	1.76	0.012	0.25	0.01
	6	1.14	0.008	0.17	0.01
Desktop	5	139	0.538	2.26	0.28
	6	62.9	0.249	1.04	0.13

Figure 5: Size of event trace in megabytes, according to the job, configuration, and frequency of invocation of the trace daemon given the use of 50,000 bytes buffers for non-X applications and 250,000 bytes buffers for X applications (desktop job).

## 4.2 Components of system overhead

In order to fully understand the impact of the different components of LTT's overhead, the following discussion is broken up along the different processing steps, from the point where the data is generated inside the kernel to the point where it is available on disk.

It is interesting to note that instrumenting the core kernel events yields an impact below 2.5%, as can be seen by comparing the sixth configuration with the first from Figure 3, regardless of the type of job run. Of course jobs that do not call upon kernel facilities as the compression example are much less disturbed by the tracing.

### 4.2.1 Kernel instrumentation

The impact of instrumenting the kernel can be seen by comparing the second configuration to the orig-



inal configuration. As the percentages show, the impact of instrumenting the kernel is very small, if not insignificant. The largest impact occurs during the desktop job and is due to the system resource intensive nature of the job. Otherwise, there does not seem to be any noticeable slowdown. Therefore, it is fair to say that using a traced kernel has minimal or no effect on the system's performance as long as the trace daemon has not instructed the trace module to record the events in its buffers.

#### 4.2.2 Trace logging

The impact of logging the events generated by the kernel into the trace module's buffers can be seen by comparing the third configuration to the second configuration. Here, the impact varies according to the type of job the system is running. In the cases where there's only one process running at all times, like the archiving and compression, the overhead is negligible. In the other cases, the kernel compilation (*gcc* and *make* do, in fact, have children) and the desktop trace, the results suggest that the copying of events from the kernel to the trace module within a kernel lock (*spin\_lock\_irq\_save*) causes scheduling contention problems.

#### 4.2.3 Trace reading

The impact of the daemon's reading the data from the trace module to its buffers can be seen by comparing the fourth configuration with the third. The overhead of the daemon copying the data is quite small but seems correlated with the overhead of the trace logging.

#### 4.2.4 Trace committing

The impact of the daemon writing the trace data to disk can be seen by comparing the fifth configuration with the fourth. Jobs where many processes contend for scheduling suffer the most significant overhead since they have to share the CPU with the trace daemon which has to write large quantities of data to file. The compilation involves several temporary files which are erased before they actually hit the disk, and is thus more CPU intensive than disk bound. The archiving job sees its overhead increase because it now has to contend with

another task that not only needs to be scheduled, but also uses the same resource, the disk.

### 4.3 Space overhead

When compiled with trace support, a modified kernel increases in size by 4KB. This includes the trace module and the modifications to the kernel. Given the current resources, this increase does not cause any problem. When tracing is activated, the configurable double buffer space is added.

The amount of disk space used by the generated traces, as can be seen in Figure 5, varies depending on the job run and can be quite large. The size of the generated trace is proportional to the number of events occurring. For CPU intensive jobs (compression), the results show that the amount of data generated is minimal. For jobs that use operating system resources intensively but are not in graphical mode (compilation and archiving), the quantity of data generated is below 10 MB per minute (0.167 MB/s).

The largest traces are generated when in the graphical environment. Here, the amount of data generated per minute is above 30 MB (0.500 MB/s) in the worst case. This is due to the constant interactions between the different applications required for the graphical display through the X server. When only core kernel events are logged, the quantity of data generated decreases to 15 MB (0.249 MB/s) per minute. Therefore 50% of the trace is composed of detailed non-core events.

### 4.4 Discussion

As seen in the previous section, some aspects of LTT use significant system resources, most importantly disk accesses and disk space. In order to reduce the overhead caused by reading the trace and writing it to disk, several mechanisms may be examined. One solution would be to take advantage of the *mmap* system call available in Linux. This would enable the daemon to map the memory used by the trace module's buffers directly in its address space and, thus, avoid any data copying from kernel space to user space by feeding the mapped buffer as the input for the *write* system call, in order to commit the trace to file. Another solution would be to feed the traces directly to the file system or to the block device driver (disk driver) from within the kernel

without using a daemon. This, though, raises other issues which go beyond the scope of our work.

Trace size reduction both reduces disk accesses and disk space requirements simultaneously. The types and structures used to record the traced events have already been optimized for size. Nonetheless, sophisticated compression algorithms use trace regularity to achieve significant size reductions at the cost of some CPU time. Using available compression tools, a compression ratio of approximately 10 was obtained on the sample traces generated. This means that the 30MB/minute trace obtained in Figure 5 could be significantly reduced to approximately 3MB/minute. Tests would need to be run to determine the cost in CPU time of the chosen compression method.

## 5 Toolkit usage and comparison

This section covers the usage of LTT in characterizing system behavior. Moreover, LTT's results are compared to the results given by conventional Unix tools.

First, section 5.1 presents example traces generated and analyzed using LTT. Then, sections 5.2 and 5.3 present real case studies where LTT has been used in order to reconstruct a system's dynamic behavior and understand its performance. Last, section 5.4 compares LTT's capabilities with that of conventional Unix tools.

### 5.1 Trace examples

In order to illustrate the type of data LTT generates and, inherently, the reason why it obtains its level of detail and accuracy, Figure 6 presents a sample trace where a process can be seen waiting to output to the hard disk. The first column presents the event type, the second the moment at which the event occurred<sup>4</sup>, the third the PID of the process running when the event occurred and the fourth the details of the event<sup>5</sup>. The sequence of events is simple, the process tries to write to a file but has to wait for I/O. It is unscheduled until the hard disk is ready. Thereafter, a verification is made to make sure no more waiting is necessary and control is returned to

<sup>4</sup>This usually consists of a seconds and microseconds pair but due to space constraints only the microseconds are presented here.

<sup>5</sup>Not all the details are given due to space constraints.

Syscall entry	(678777)	1021	SYSCALL : write
File system	(678779)	1021	WRITE : 3
File system	(679107)	1021	START I/O WAIT
Sched change	(679151)	0	IN : 0; OUT : 1021
...			
IRQ entry	(691806)	0	IRQ : 14, IN-KERNEL
Process	(691818)	0	WAKEUP PID : 1021
IRQ exit	(691823)	0	
Sched change	(691824)	1021	IN : 1021; OUT : 0
File system	(691826)	1021	START I/O WAIT
File system	(691827)	1021	END I/O WAIT
Syscall exit	(691936)	1021	
Syscall entry	(691941)	1021	SYSCALL : sigreturn

Figure 6: Sample trace: Process waits for I/O.

the process.

Though the example is simplistic, it demonstrates the level of detail attainable using LTT. *gdb*, for instance, would have been fairly inadequate in helping us to figure out this sequence of events.

Figure 7 shows an example trace graph drawn by LTT. At the top of the window, we can see a menu and a toolbar. Three thumbnails are used to present the data about the trace. The first presents the graph. The second contains performance data. The last contains the raw list of events. The graph thumbnail is separated in two parts. The left side holds a list of all the processes that were active during the trace. The right side holds a graph that shows the flow of events in time. Vertical lines mark a transition in control, whereas horizontal lines signify time flow. The graph in Figure 7 shows how *minilogd* got scheduled right after the system clock. *minilogd* then did a *newstat* system call. The kernel did some work and gave control back to *minilogd* which then called on *poll*. The kernel did not find anything for *minilogd* and, consequently, scheduled the idle task since no other task needed the CPU.

### 5.2 Characterizing a normal workstation

In order to characterize a normal workstation, a set of applications must be chosen to represent the common usage of a system as a workstation. Thereafter, measures can be made using LTT and using conventional means. The accuracy of the conventional tools can then be assessed by comparing to the more accurate detailed data available in the trace.

The following applications were monitored for a period of 30 seconds: *X server*, *netscape*, *staroffice*, *x11amp* and a script running *ps* every 5 seconds.

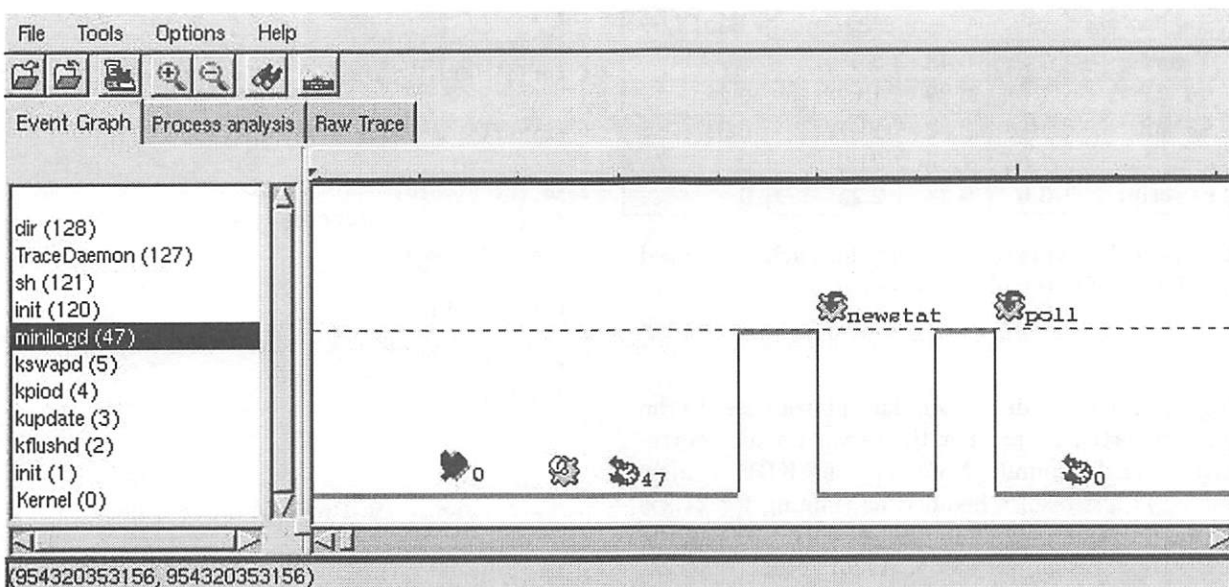


Figure 7: An example trace graph.

Application	<i>ps</i>	User	Running	Wait I/O
<i>X server</i>	8.6	12.79	15.28	0.04
<i>netscape</i>	3.25	15.43	17.17	1.85
<i>staroffice</i>	2.7	4.73	5.52	2.60
<i>x11amp</i>	1.15	1.62	2.19	0
<i>ps script</i>	0.43	0.04	0.08	0

Figure 8: Percentage of activity for each monitored application for the observed workstation.

During that period of time, LTT's trace daemon was recording events and *ps* was called upon every 5 seconds. The applications were used in a typical fashion. *netscape* was used to view 2 web sites. A document was being modified in *staroffice*. *x11amp* was playing an mp3 file. Figure 8 presents the results of the 30 seconds run. Column *ps* contains the average value of the percentages of CPU usage reported by the many runs of the *ps* utility during the test. Column *User* shows the percentage of the time the system was executing actual application code in user mode, as reported by LTT. Column *Running* shows the percentage of time the said application was scheduled, as reported by LTT. Column *Wait I/O* shows the percentage of time during which the said application was waiting for I/O.

As the results show, the conventional monitoring tool, in this case *ps*, fails to provide the observer with an accurate appreciation of the system's be-

havior. In fact, *ps* reports that *netscape*'s CPU usage was 3.25% whereas the real figure is 15.43%, misleading an observer into believing that the said application's CPU usage is more than 10% lower than its actual value. Moreover, there is no way to find out the amount of time during which the process is not running because it does not need to run or the amount of time during which it is waiting on an I/O resource. Also, it is important to know how much time was spent executing kernel code and how much time the system was idle. For the workstation, 59.06% was spent in kernel mode and the idle task was scheduled for 49.31% of the sample time.

### 5.3 Characterizing a small server

Here a small FTP server is set up to provide service to a client connected through Parallel Port IP using a null modem printer cable. The set of applications observed is *X server*, *ftp daemon*, KDE window manager *kwm*, and a script running *ps* every 5 seconds. The client requests 2 files for download from the server and the operation is monitored for 60 seconds. Figure 9 presents the results for the run. Note that there are 2 ftp entities, one for each copy of the daemon serving a corresponding file download.

As for the workstation, the results show that *ps* does not provide a clear profile of the observed system. Whereas for the workstation the time spent execut-



Application	ps	User	Running	Wait I/O
<i>X server</i>	7.2	0.48	3.45	0
<i>ftp #1</i>	13.1	0.29	10.92	0.26
<i>ftp #2</i>	10.8	0.32	11.77	0.63
<i>kwm</i>	1.63	1.33	21.2	0
<i>ps script</i>	3.6	0.38	2.25	0

Figure 9: Percentage of activity for each monitored application for the observed server.

ing application code was similar in most cases to the time reported by *ps*. For the server no such correlation can be found. Note that the KDE window manager has been scheduled as running for 21.2% of the duration of the sample, yet only 1.33% of the CPU time was in user mode. As for kernel code execution and time spent idle, the server spent 94.71% of its CPU time in kernel mode and the idle task was scheduled for 18.72% of the sample time.

#### 5.4 Conventional tools comparison

The main interest of LTT being the fact that it provides information previously unavailable, it is important to compare LTT to existing analysis software. To this end, the following paragraphs compare LTT to *gdb*, *ps*, *gprof* and *time*.

Section 5.1 presented a simple trace example that showed that *gdb* was an inadequate tool in some circumstances. A more blatant example of the limits of *gdb* would be to try using it to figure out synchronization problems. Because it modifies an application's behavior, using the *ptrace* system call, it is often impossible to use *gdb* to reproduce synchronization problems, much less debug them. Moreover, synchronization problems, depending on their nature, can disappear when debugged using *gdb*. In the case of simple interactions, the crude workaround of inserting *printf*s in the debugged code is usually sufficient to help the developer solve these types of problems. But in complex software systems, this workaround is seldom adequate. In this regard, LTT fixes this type of problem by providing the developer with the exact sequence of events as they occurred on a live system. Thereafter, tracking a synchronization problem amounts to tracing the conflicting events such as IPC and socket communication and analyzing the sequence of their occurrences. Furthermore, LTT does not modify the behavior of the observed system since

```
int i, j;
void fct_A(void)
{ for(i = 0, j = 0; j < 10000000; j++)
  i++;
  printf("i's value : %d \n", i);
}
void fct_B(void)
{ for(j = 0; j < 1000000; j++)
  sched_yield();
}
int main(void)
{
  fct_A();
  fct_B();
}
```

Figure 10: Profiled source code.

events are logged in the sequence of their occurrence and the locks held to record those events are held for a very short time.

As has been demonstrated above, LTT is also very helpful in figuring out performance issues regarding an observed system. More importantly, compared with the performance data generated by conventional tools, the performance data generated by LTT matches more closely the actual behavior of the observed system of process. This has been demonstrated for *ps*. It is important to note that though *ps* is not the only tool used for performance measurement on modern Unix systems, and certainly not the most precise one, it is by far the most important because of its wide-spread usage and adoption as a legitimate way of quantifying a system's performance.

Another tool commonly used to measure performance is *gprof*. Contrary to *ps*, it is usually used to analyze the behavior of a single application. Here again, the data provided is statistical at best. To illustrate this, Figure 10 presents a portion of code that was profiled using *gprof*.

Using *gprof* we learn that the application spent a total of 250ms executing. 170ms were spent in *fct\_A* and the rest, 80ms, were spent in *fct\_B*. Using LTT, we find that the application actually spent 3.28s scheduled and that 2.34s were spent executing code belonging to the application. The rest of the time was spent running system code for the application. It is interesting to note that LTT reports that the application spent 972ms in cumulative calls to *sched\_yield*. This has gone unnoticed by

*gprof*, which simply reports that 80ms were spent in *fcntl.B*. Moreover, *time* corroborates LTT's results and reports that the application ran for 3.64s. *time*, though, reports that 2.71s were spent running system code and 0.61s were spent running user code. These times are calculated using the statics cumulated by the sampling done in the kernel. In essence, these results are similar to results reported by *ps*, which are known to lack precision.

The difference between the results given by LTT and the results given by *gprof* are due to the difference in the way data is acquired. *gprof* uses the system clock to sample the process' behavior. With LTT, we can see the timers used by *gprof* to sample the code going off and generating *sigreturns* once they are done sampling. By observing the trace, we can see that the profiling timeouts very often occur within a call to *sched.yield*. Therefore, the system is running kernel code at that time and the time from that last sample is attributed to the system. This is why *time* attributes the wrong values to the different components of performance and it is the reason why *gprof* reports incorrect values. The same experiment was run using *gettimeofday* instead of *sched.yield* and gave very similar results.

## 6 Future directions

LTT has been available for some time through its home page (<http://www.opersys.com/LTT>). Many users have already used it for tracing and profiling purposes. In providing this project through the GPL license, the authors hope that it will benefit as many users as possible while offering advanced functionality.

The extensibility of LTT is provided by its openness and its modularity. Adding events to the list of events already being traced amounts to adding an identifier in the source code and placing the required trace instruction in the corresponding place in the kernel source code. Simplifying this process, an extension is currently being developed that enables the dynamic creation of event IDs and their automatic recognition by the trace analysis software. This will eliminate the need to modify LTT in any way to add traced events.

As said before, LTT comes with a versatile data analysis and presentation software tool. The later enables the user to view the event trace in a browsable graphical form. The trace is presented as a

control graph where changes of control from/to the kernel are easily seen. It also provides a command-line interface enabling the user to access all of its functionality without requiring a resource intensive GUI.

There are many interesting future research avenues. For instance, given the precision of LTT's results, it would be interesting to implement a quality of service oriented kernel resource management that would use trace analysis feedback in making its future decisions. This would enable system administrators to fix quotas on the usage of most system resources.

Given LTT's ability to track detailed kernel events, it can easily be used as a component of security auditing and watchdog tools. Events matching a certain description could trigger logging or the execution of a program. Another usage would be to create a graph for tracking some of the monitored events. Rather than polling the content of */proc*, this graphing tool would be fed directly by the trace module, increasing the precision of the data. It could resemble the Windows NT Performance Monitor, though the underlying functionality would differ greatly.

Merging and interfacing with related open source projects is important to the authors. Prime and foremost, this includes eventually integrating the kernel patches to the main Linux kernel source tree. This would bring built-in tracing capabilities to every Linux user.

## 7 Conclusion

In this paper, we have presented a novel way of recording and analyzing system behavior. Our results have shown that LTT's overhead is minimal and that it provides unique data sets. These data sets have successfully been used to reconstruct the dynamic behavior of systems. The relatively low accuracy of conventional system monitoring tools was also shown, thus motivating the use of kernel tracing facilities, whenever a precise characterization is required.

The tools developed as part of the Linux Trace Toolkit are modular, extensible and openly available, making it easy to extend and customize them. Such tools will be crucial to the development of future computer systems due to the ever-increasing complexity of the software and hardware developed.

## References

- [1] Rational Software's Quantify, <http://www.rational.com/products/quantify>.
- [2] WindRiver's WindView, <http://www.windriver.com/products/html/windview2.html>.
- [3] QNX's DejaView, <http://www.qnx.com>.
- [4] Nematron's HyperKernel Trace Utility, <http://www.nematron.com/solutions/software/hyperkernel/hyperkernel.html>.
- [5] GNU General Public License version 2, <http://www.gnu.org/copyleft/gpl.html>.
- [6] G. Ammons, T. Ball, and J. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, 1997.
- [7] M. Bach. *The Design of the Unix Operating System*. Prentice Hall, 1986.
- [8] T. Ball and J. Larus. Efficient Path Profiling. In *Proceeding of MICRO-29*, 1996.
- [9] J. Anderson et al. Continuous Profiling: Where Have All the Cycles Gone. In *16th ACM Symposium on Operating Systems Principles*, 1997.
- [10] M. Beck et al. *Linux Kernel Internals: Second Edition*. Addison-Wesley, 1998.
- [11] X. Zhang et al. System Support for Automatic Profiling and Optimization. In *16th ACM Symposium on Operating Systems Principles*, 1997.
- [12] S. Graham, P. Kessler, and M. McKusick. gprof: A call graph execution profiler. In *SIGPLAN Notices*, 1982.
- [13] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. In *ACM Transactions on Modeling and Computer Simulation*, volume 7, pages 78–103, January 1997.
- [14] A. Rubini. *Linux Device Drivers*. O'Reilly, 1998.
- [15] David A. Salomon. *Inside Windows NT: Second Edition*. Microsoft Press, 1998.
- [16] Uresh Vahalia. *Unix Internals: The New Frontiers*. Prentice Hall, 1996.

# Pandora: A Flexible Network Monitoring Platform

Simon Patarin and Mesaac Makpangou

*INRIA SOR Group*

*Rocquencourt, France*

Simon.Patarin@inria.fr, Mesaac.Makpangou@inria.fr

## Abstract

This paper presents Pandora, a network monitoring platform that captures packets using purely passive techniques. Pandora addresses current needs for improving Internet middleware and infrastructure by providing both in-depth understanding of network usage and metrics to compare existing protocols. Pandora is flexible and easy to use and deploy. The elementary monitoring tasks are encapsulated as independent entities we call monitoring components. The actual packet analysis is performed by stacking the appropriate components. Pandora also preserves user privacy by allowing control of the “anonymization” policy. Finally, the evaluation we conducted shows that overheads due to Pandora’s flexibility do not significantly affect performance. Pandora is fully functional and has already been used to collect Web traffic traces at INRIA Rocquencourt.

## 1 Introduction

Network monitoring is essential for the improvement of Internet performance. It serves to capture usage profiles, to evaluate the impact of Internet services (e.g. replication and cooperative caching), to compare the overheads of different implementations, and to help debug complex distributed applications.

In recent years, several network monitoring tools have been proposed. These include naturally `tcpdump` [11], and specialized software like BLT [2] and `HttpFilt` [21] for HTTP extraction, and `mmdump` [1] for multimedia monitoring. One must also mention more generic platforms like IPSE [6] and Windmill [12].

As the Internet grows, the demand for a flexible monitoring system increases. Such a system can be used to ensure good performance while keeping pace with the rapid evolution of Internet protocols and services by enabling rapid implementation of dedicated monitoring tools.

Unfortunately, existing network monitoring tools remain too specific. They are designed to collect information required for some particular analysis. They often depend on a particular version of a protocol, or a particular configuration of the underlying in-

frastructure. For instance, `HttpFilt` only works for HTTP/1.0. IPSE assumes the existence of a gateway where one observes the entire traffic within the monitored organization. Such dependencies make most existing tools redundant if the targeted protocol evolves or the assumed configuration changes. Also, it makes them difficult to adapt and cope with new problems.

This paper presents Pandora,<sup>1</sup> a flexible and extensible network monitoring platform that can be easily adapted to monitor new Internet protocols or application-specific ones, while still offering good performance. Pandora uses passive network monitoring to reconstruct high level protocols while keeping track of lower level events. It provides basic building blocks implementing the commonly used analysis.

The rest of the paper is organized as follows: Section 2 presents our design goals for Pandora. Section 3 describes the architecture of the system, while Section 4 describes how it is realized. Then, Section 5 focuses on the implementation. Next, we consider two examples of use of Pandora in Section 6; then we discuss the performance of the system in Section 7. Finally, we compare Pandora to related work in Section 8 and present some concluding re-

<sup>1</sup>Our platform is called Pandora to recall the potential dangers of overly intrusive curiosity.



marks in Section 9.

## 2 Design Goals

Our design has four main goals: monitoring a system should not affect the system's behavior; the tool should be fast enough to monitor high-bandwidth links; user privacy must be preserved; and the tool must be flexible and simple enough to allow reuse in diverse applications.

### 2.1 Preserving the Quality of Service

A monitoring tool should perform its work without being noticed by the system or its users. In particular, it should not introduce artificial perturbations into the environment. It should also not degrade the quality of service provided to the users of the system.

Although easier to realize, we decided not to implement the tool as an active element of the system: a proxy, for example, could be used to intercept the traffic and to log information. However, such an active tool necessarily has an impact on the system's performance; in the case of a failure, for example, the monitored service could be interrupted.

Therefore, we decided to use passive network monitoring. Such a tool captures the packets on the network and treats them without interfering with the actual traffic. This choice implies in particular that we must be able to deal with packet losses, without the opportunity to request the sender for packet re-transmission.

### 2.2 Performance Issues

A monitoring tool can be used to trigger a fast reaction to certain events. For example, one could decide to modify a Web cache's configuration because of a decrease in the observed quality of service. Therefore, the monitoring tool must be able to process all packets in real-time; this means that it cannot be designed only to store packets and process them off-line.

Our system must be able to monitor medium-speed links, such as a 100 Mb/s Ethernet and a T3 wide-area link. Therefore, the design of the tool must be light enough to accommodate such a traffic. Moreover, it should be able to deal with load peaks which can occur frequently in networking systems.

### 2.3 Preserving User Privacy

A serious concern when monitoring a system is to preserve user privacy. Therefore, we should not output personal information such as which Web pages a particular user has accessed. Yet, to have enough insight into the system's behavior to provide interesting results, we often need precise information about user behavior.

We consider that there must be a tradeoff between user privacy and the level of details that are provided by the monitoring tool. Depending on the planned use of the collected information, different levels of trace "anonymization" must be used.

Therefore we consider that privacy should be treated as a policy (hence flexible) and that the system should only enforce a level of privacy compatible with the study to be done.

### 2.4 Ease of Use and Deployment

We anticipate that the monitoring system will be used for various purposes: gathering traffic traces for several Internet protocols (e.g. HTTP, DNS or ICP), comparing network overheads for different protocols stacks, or debugging distributed applications. This list of possible uses of the system is of course not exhaustive.

An administrator should be able to use the same monitoring tool for numerous different purposes easily. Therefore, the system must provide useful default options while being easily customizable. Moreover, it should not require specific hardware, and it should be portable across several, widely used, platforms. Finally, new protocols arise every day that people might want to analyze. Also, it should be easy to add new protocol-specific elements to the system.



### 3 Architecture

Pandora is designed as a stack of monitoring components. Each component encapsulates an elementary monitoring task (e.g. IP layer reassembly, TCP layer resequencing, etc.). We first illustrate the problem and its solution with the example of HTTP monitoring; then, we describe the general design of the platform.

#### 3.1 Example of HTTP Extraction

This example consists of gathering the Web traffic generated by a given user community. Such traces can lead to a better understanding of traffic patterns, and hence to the design of better protocols and tools.

The protocol we need to monitor here is HTTP/1.1 [5]. HTTP follows a request/response model. Meta-data carried by headers are line-oriented and use textual attributes and values. Version 1.1 of the protocol introduces the possibility of making persistent connections between clients and servers and of pipelining requests and responses (not waiting for the other endpoint to reply before sending additional data on the same connection).

The `libpcap` [10] library is used to capture any packet which passes through the network. It provides raw network packets, i.e. arrays of bytes. To extract the useful information from such packets, we need to reconstruct the HTTP sessions. First, we remove the link layer headers (the Ethernet headers, for example). This provides an IP packet. IP packets can be fragmented when traveling through the networks. Therefore, we need to reassemble the fragments (based on the IP headers). Once the reassembling is done, we can extract a TCP packet. Based on the TCP headers, we can determine which TCP stream it is part of, its proper place in the stream, etc. After demultiplexing and ordering TCP packets, we obtain the HTTP stream. We finally have to parse the HTTP header fields in order to obtain the HTTP meta-data. Finally, the request meta-data must be matched with the corresponding response meta-data before being output.

#### 3.2 Basic Components

As one may notice from the above example, a few elementary tasks appear, namely: IP reassembly, TCP resequencing and HTTP request/response matching. The only dependence between those tasks is the order in which they are performed. These tasks operate on packet *flows* rather than on individual packets. A packet flow is a sequence of packets related to the same higher-level entity (an IP packet for IP fragments, or a connection for TCP packets for example) exhibiting *temporal locality*.

The notion of temporal locality depends on the nature of the flow itself, and can be seen as a threshold beyond which the flow is considered to be closed (a similar but more restrictive definition is given in [4]). In our example, a packet flow could be the set of all fragments — including duplicated ones — forming a single IP packet, or the set of all TCP packets sent from a browser to a Web server, containing HTTP requests.

To capture the separate tasks that must be performed to produce the trace, and the order in which they must be proceeded, we define two basic notions: monitoring components and component stacks. A monitoring component (or simply, component) is responsible for a specific elementary task, while the stack is the structure where these components are chained together.

A component may be considered as an operator on packet flows: it takes some flow as input, performs its work on it and then produces a flow of a different nature as output. Thus a component designed to perform IP reassembly transforms a flow of IP fragments into a flow of IP packets.

A component relies only on the properties of its input. In particular, it does not have to know about the other components in its stack. This lets us replace a component by an equivalent one, or to introduce new components into the stack with no effect on the rest of the stack. For example, imagine that we want to encrypt the data collected (a perfectly legitimate issue); we have only to add a component that will encrypt IP addresses and URLs, after the HTTP protocol has been parsed.

Many different monitoring experiments can be conducted without effort, if all the required components exist, or at minimal cost concerning only the de-

velopment of the missing ones. Using components makes it easier to debug the monitoring path; effort can be concentrated on the optimization of components that constitute bottlenecks.

The stack determines how components are chained. It represents an ordered set of components through which packets flow from one end to the other. It is important to notice that, in such a structure, each component has exactly one input and one output; in other words, there is a single, linear data flow for the entire stack. Such a simple stack model has two drawbacks. First, packets that belong to distinct connections are merged into an unique flow. This implies that every components that operates on separate packet flows will have to perform the demultiplexing on its own, since having only a single output is equivalent to re-multiplexing everything before passing packets to the next component. With respect to our example, packets coming from all HTTP connections are captured and delivered to the system interleaved, and demultiplexing must occur at IP, TCP and HTTP level. Second, this simple stack model (with a unique data flow) makes it impossible to shortcut some of the processing components, even if we know *a priori* that this will not be necessary: all packets must pass through each component. For example, we stated that IP packets could be fragmented, yet rather few are. Given that this property can be determined early (by simply looking at IP flags), why should we make them all pass through the reassembly components (demultiplexing and effective reassembly)?

These may look like straight-forward, standard problems, but it requires some careful thought to fit them easily into the component framework.

### 3.3 Generalized Stacks

The two limitations of the simple stack model we mentioned above have led us to extend this model by introducing *control* components. They allow several data flows to coexist in a single stack. This approach permits us to save resources and avoids unnecessary component traversals. Furthermore, taking flow control mechanisms out of processing components permits component developers to concentrate their efforts on the precise functionality they want to implement.

The control components are the following:

**Switch component:** It permits packets to follow along different processing paths: configured with a fixed number of alternative paths, it can forward a packet to any of them, according to some hard-coded internal logic. In our case-study, it may be used with the IP re-assembly component: routing IP fragments to the reassembly sub-stack but routing complete packets directly to TCP packet extraction component.

**Demux component:** Such a component dynamically instantiates a dedicated sub-stack for each new packet flow it detects. It then forwards the packets to their relevant stack. Once a session is finished (e.g. a TCP connection is closed), the sub-stack is removed.

Figure 1 shows one solution for the Web trace collection example that uses switch and demux components. The "connection demux" receives TCP packets from the IP to TCP extraction component. It identifies which flow each packet belongs to. For each flow, it dynamically creates a "direction demux" which in turn identifies in which direction of the flow each packet is going (from the client to the server, or the other way round). TCP packets are then passed to components performing TCP resequencing and HTTP reconstruction. Finally, the two opposite HTTP streams (i.e. a stream of requests and the corresponding stream of responses) are given to the HTTP matching component, which determines which request corresponds to which response.

### 3.4 Benefits of the Stacking Approach

**Flexibility:** One can easily replace one monitoring component by an alternative implementation to adapt to a specific situation, or to test new algorithms.

**Evolvability:** Protocols are evolving rapidly and independently from each others. The use of components permits easy adaptation to these changes.

**Extensibility:** It is straight-forward to add new protocol extraction capability, just by adding a new component to an existing (lower-level) stack.

**Modularity:** Each task is encapsulated in its own component which enforces a clear division between mechanism and policy, eases readability

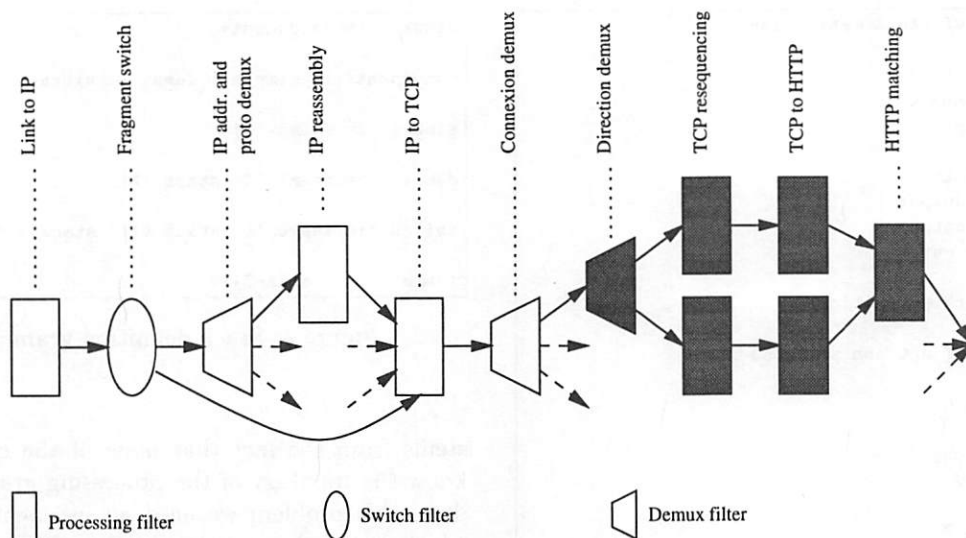


Figure 1: The component stack used in the Web traffic collection example. The shaded components are those dynamically created by the “connexion demux” when a new TCP flow is identified.

and has proven very valuable for maintaining existing code and debugging. There are no obscure side effects, nor hidden dependencies.

### 3.5 Configuration

Figure 2 presents a sample configuration file for an application that logs packets in distinct log files according to their transport layer protocol (TCP, UDP or ICMP). In details: packets first flow through the `ipfragswitch` switch component. This component makes IP fragments go into the IP reassembly sub-stack and let the other not-fragmented packets skip this part. Inside the IP reassembly sub-stack, packets are demultiplexed according to their IP addresses, protocol number and identifier in the `ipfragdemux` demux component. Then the `ipreass` component reassembles demultiplexed IP fragments. The timeout specified in the corresponding option section tells the filter to flush any incomplete packet after a 60 seconds inactivity period. At this point, all IP packets reaching the `ipproto` component are complete IP packets. This switch component forwards them to distinct sub-stacks according to their transport protocol. The options it is given tells it what are the actual branch indexes used. TCP packets go to the first one where they are parsed and written to a log file. As are UDP and ICMP packets to the second and the third sub-stack, respectively. Finally, all packets with a different protocol

skip this and are discarded.

The configuration file is made of several sections. The stack section (beginning with a `[stack]` tag) describes the components used and how they are chained. We designed a small language to specify this. Its grammar is presented in Figure 3. A stack is composed of a number of components. The simple components are identified only with their name.<sup>2</sup> The switch and demux components are described by their name as well as the definition of their sub-stacks: the demux is given the definition of the sub-stack to instantiate when identifying a new session; the switch is given the list of sub-stacks to which it can forward packets.

Other sections of the configuration file allow us to set up options for any component mentioned in the stack section. Such option sections start with the name of the component enclosed in square brackets: for example, `[ipfragdemux]`. In addition, components of the same type are numbered from left to right, starting at 0, in order of appearance. This is needed in cases where several instances of the same component are used, and one wants to set different options for each instance. That is, one creates an option section identified by this component name followed by its rank (e.g. `[output(0)]`). Unnum-

<sup>2</sup>The “-” connector is optional and is used only for clarity (e.g. one can alternatively use `component1 - component2` or `component1 component2`).

```

# beginning of stack definition
[stack]
ipfragswitch (
    ipfragdemux <
        ipreass
    >
) ipprotoswitch (
    tcpscan output |
    udpscan output |
    icmpscan output
) discard
# end of stack definition

# beginning of options sections
[ipreass]
timeout = 60

[ipprotoswitch]
tcpbranch = 0
udpbranch = 1
icmpbranch = 2

[output(0)]
static file = tcp_pkts.dat

[output(1)]
static file = udp_pkts.dat

[output(2)]
static file = icmp_pkts.dat
# end of option sections

```

Figure 2: Configuration file for a stack that reassembles fragmented packets and logs TCP, UDP and ICMP packet into different files. Packets of other transport level protocols are discarded.

bered option sections apply to all components of the specified type. Inside an option section, each line describes the setting of an option according to the following syntax:

**option\_name** = **option\_value**

Values can be of numeric or string type. String type values are processed by a function defined by the component developer. By default, this function is evaluated each time the component is created. Yet, this may not be always suitable; prefixing option name with the keyword **static** guarantees that the function will only be evaluated once.

## 4 Stack Instantiation

Once the stack and component options have been parsed, one has to instantiate the stack. This involves dynamically building the sequence of components and linking them appropriately. The difficulty

```

stack ::= component+

component ::= simple | demux | switch

simple ::= fname '-'?

demux ::= fname '>' stack '>>'

switch ::= fname '(' stack '|' stack)* '>>'

fname ::= [a-zA-Z]+

```

Figure 3: Stack definition grammar.

stems from the fact that none of the components know the topology of the processing graph. To address this problem we need an independent entity which holds this graph and the configuration parameters to be passed to different instances of components. We call this entity the *dispatcher*.

### 4.1 Component Creation

The dispatcher is responsible for creating components — which includes properly setting any dynamic options it might need, and building the ordered sequence of components specified in the stack description.

When created, a component does not know anything about other components in the stack. If it wants to pass a packet to its successor, it must first ask the dispatcher for a pointer to its successor (which it stores for future accesses). If the requested component does not exist, the dispatcher instantiates it, then returns its reference to the calling component. This is the case for simple components: successors are never created until they are accessed. This way, the dispatcher does not need to keep track of previously created components.

However, this technique poses problems for demux and switch components: the ending components of every created branches must share the same successor (the *multiplexing* component). To address this issue, we note that there is a one-to-one mapping between the demultiplexing and the multiplexing, and that demultiplexing always occurs before multiplexing in the graph. It is therefore sufficient to create the multiplexing component along with the demultiplexing one (be it demux or switch). Then, each component in a demultiplexed branch carries a reference back to its demultiplexing component



(this reference is passed dynamically when the component is created). When the last component in a branch asks the dispatcher for its successor, the dispatcher retrieves the reference to the multiplexing component from the demultiplexing one, by following a back-pointer to the branch point.

## 4.2 Memory Management

As with their creation, destruction of components is performed incrementally. The process starts with a component notifying the dispatcher of its intention to destroy its successor (or one of them, if this is a demultiplexing component). This notification is then propagated by the dispatcher. It first asks the target component to prepare for destruction. Upon completion of this request, the dispatcher effectively destroys the component (and resets the caller's pointer to a nil value). The process iterates up to the component at which the initiator's branch is multiplexed, where it stops. For a multiplexing component, destruction preparation involves recursively initiating a new destruction process for each of its branches. Other components simply flush all their unprocessed packets.

We have two ways to trigger component collection: active and passive. Active collection occurs when a component determines autonomously that its processing is finished. It then notifies its predecessor<sup>3</sup> that it can start the destruction process. This is the case when an IP reassembly component receives the last fragment of an IP packet. Passive collection occurs for inactive components whose processing remains unfinished either because the component has no way to know it, or because of a packet drop. This process is based on predetermined timeouts.

Pandora provides a generic framework to timeout inactive components. A global clock is maintained according to the time-stamps of incoming packets. Then, each component has the possibility to register itself for a specific timeout. When no packets are passed to the component during the specified period, the destruction process for that component (and all its successors) is triggered.

It must be noted that the choice of timeout value is the result of a delicate tradeoff: too short a timeout may interrupt ongoing connections, while large

<sup>3</sup>By way of the return value of the function used to pass a packet to a successor component, or via the dispatcher.

timeouts might dramatically increase the memory footprint of the application. There is no canonical value: it depends on the nature of the observed traffic and of the type of link being monitored.

## 4.3 Concurrent Processing

Pandora allows components to be executed concurrently insofar as the logical flow of packets is respected. More precisely, it offers the possibility of running some parts of a stack in different threads on the same machine or even on distinct machines, forwarding packets over the network.

We could, for example, allocate a thread to fetch packets, to limit the risk of kernel queue overflow (for kernel structure has a fixed size, whereas our buffers are only limited by the available memory). It is also possible to perform some kind of load balancing or to correlate several observations made from different vantage points in cases where information is not fully available at a single location. For example we may think of HTTP requests and responses flowing through two distinct links. Two distinct kinds of components may be used to achieve this: thread and I/O components.

Each thread component creates a new thread of execution in the component stack. All packets added to such a component are forwarded to the newly created thread. In other words, they split the stack into two parts, each of them being executed in an independent thread. A thread component manages synchronization — packets are added in a shared, mutex-protected, FIFO — but relies on other components being fully reentrant.

I/O components are used to exchange packets across the network. A TCP connection is established between an output component (acting as a client) and an input component (acting as a server, thus allowing packets coming from different machines to be merged). Input components dispatch incoming packets to the remainder of the stack. An administration facility allows easy use of these components: it sets up the TCP server, performs the necessary configuration on the client side of the connection (setting the name and incoming port of the remote host) and monitors clients for crashes, restarting them if necessary (although no state recovery is performed).

## 5 Implementation

The software consists currently of about 15,000 lines of C++ code. Beyond the core system, described Section 4, we have implemented various monitoring components — grouped into a library.

The list of components included in this library is presented in Table 1. The library also provides all data structures by these components (packet types, hash functions, etc.).

## 6 Examples of Use

Pandora has been used in two different, but related, fields: Squid cache and Web traffic collection. These two examples exercise the different features of Pandora presented so far, and its ability to handle transactional protocols. The cache monitoring example was set up in very short amount of time (it took only a few hours, starting from scratch).

### 6.1 Cache Monitoring

The Internet Cache Protocol [19] permits caching proxies to cooperate (in particular Squid [18, 20] caches).

Basically, for each miss, a cache emits ICP queries to know if one of its siblings possess the requested document. Each peer responds in turn with a hit, a miss, or an error message. In a second phase, the original cache forwards the request either to a sibling that responded with a hit, or to the original server. Each message is distinguished by a unique 32-bit identifier and uses UDP as the transport protocol.

Figure 4 shows the stack we use to evaluate the overheads caused by this protocol in terms of additional delay and bandwidth usage.

The first part of the stack (up to `ipcnxdemux`) is dedicated to IP reassembly and has the same structure we discussed in Section 3. The `ipcnxdemux` component demultiplexes packet according to their source and destination IP addresses, but no discrimination is made on the direction of the connection

```
[stack]
ipfragswitch (
  ipfragdemux <
  ipreass
>
) ipcnxdemux <
  scanudp - scanicp - icpdemux <
  matchicp
>
> output
```

Figure 4: Cache monitoring stack definition.

(i.e. we make no distinction between packets coming from the requesting host or from the responding host). The `scanudp` component extracts UDP packets from IP ones and passes them to `scanicp` component. The latter in turn produces ICP packets. These are demultiplexed according to their request identifier in the `icpdemux` component. Then, given our demultiplexing steps, only the two matching ICP messages are finally passed to the `matchicp` component, whose only work is to associate them in a single ICP transaction packet.

### 6.2 Web Traffic Collection

We already introduced HTTP extraction in Section 3. This application<sup>4</sup> is meant to help to characterize the Web activity of a community of users. This information can then be used to dimension cache infrastructure for example (as in Saperlipopette! [15]), or to give hints about which sites are worth mirroring.

Figure 5 shows the configuration file of the Web traffic collection system.

```
[stack]
tcpscan - tcpcnxdemux <
  tcpdirdemux <
    tcpresq - httpscan
  > anonymize - httpmatch
> output
```

Figure 5: Web traffic collection stack definition.

The `tcpcnxdemux` component demultiplexes packets according to their connection identifier (the 4-tuple IP address and port for the source and the destination) independent of the direction of the flow. The

<sup>4</sup>A Web traffic collection experiment has actually been led at INRIA for one month using this software.



### Monitoring Components

PcapHandler	Encapsulation of the libpcap library
ICMPScan	Extracts ICMP packets from IP packets
TCPScan	Extracts TCP packet from IP packet
UDPScan	Extracts UDP packet from IP packet
ICPScan	Extracts ICP packet from UDP packet
RPCScan	Extracts RPC packet from UDP packet
DNSScan	Extracts DNS packet from UDP packet
HTTPScan	Extracts HTTP message from TCP packet flow
IPReassembly	Reassemble IP fragments into IP packet
TCPResequence	Resequence TCP packets in TCP packet flow
HTTPMatch	Matches HTTP request and response into HTTP transaction
GenMatch (template)	Generic component allowing to produce transaction packets for "simple" protocols (currently used for ICP, RPC, DNS).
Input	Passes to the first component of the stack the packets transmitted over the network between two instances of Pandora
Output	Outputs packets to log file (dealing with file rotation), console or to another instance of Pandora through the network
Discard	Discards any packet it receives
Thread	Inserts a thread into the processing path

### Demux Components

GenDemux (template)	Generic demux component (given the input packet type and the hash function to apply)
TCPPortDemux	Demultiplexes TCP packets according to their port number

### Switch Components

IPFragSwitch	Switches between fragmented and non-fragmented IP packets
IPProtoSwitch	Switches between transport level protocols of IP packets (currently TCP, UDP, ICMP)

Table 1: Existing components and packet types in Pandora.

separation of both flows is made by the `tcpdirdemux` component. Once TCP packets are resequenced, the `httpscan` components extract HTTP messages (requests and responses) from each flow. Then, these messages are multiplexed, so that requests and their corresponding responses belong to the same flow. After being anonymized, requests and responses are matched to each other (without any interference from messages of other connections).

## 7 Evaluation

Performance is a major issue that we set out to address. Indeed, packets are buffered by the system in a kernel queue which is emptied by the monitoring process. When the queue is filled up,<sup>5</sup> packets are discarded by the system. It is a common belief that flexibility has a price: this could jeopardize our performance goal. After having described our experimental environment, we will show the results

<sup>5</sup>With the BSD Packet Filter [13] on a DEC OSF1 operating system, this queue has, by default, a maximum size of 256 packets.

of several tests meant to measure respectively the overheads due to our flexible design, those related to demultiplexing and Web traffic collection. Finally we will quantify the effective throughput of Pandora when used to collect Web traffic in more realistic conditions.

### 7.1 Experimental Environment

In order to stress our system we used several traces collected by `tcpdump` on the link connecting INRIA with the outside world. The traces were read from a file on a local disk, and the results presented in the following sections correspond to the arithmetic mean of 50 runs of the same test (we always present the standard error of these measurements).

The workstation used to run these tests was a DEC *Personal Workstation* using a 21164A processor at 500 MHz, with 684 MB of RAM. We chose relatively small trace files (500000 packets) so that their entire content can remain in main memory between successive runs.

All times measured are the sum of system and user execution time (in seconds) as given by `getrusage` on this workstation (still used for light office tasks at the same time).

## 7.2 Flexibility Support Overhead

In this section we will quantify the overheads directly linked to our design: transforming layering and demultiplexing into “first-class” entities.

### 7.2.1 Layering

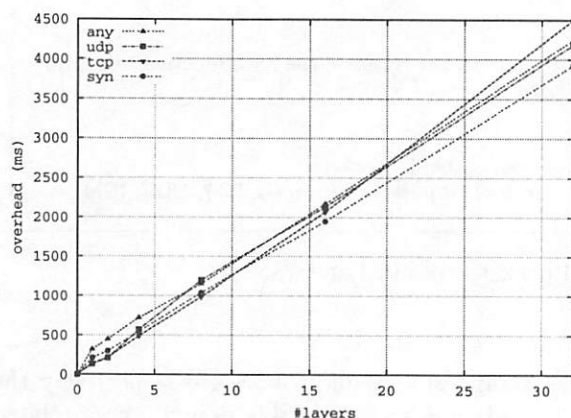


Figure 6: Evaluation of layering overhead, using the ANY, SYN, TCP and UDP traces.

To measure the cost of layering we ran Pandora against four traces (whose characteristics are presented in Table 2) with a variable number of identical components that are only passing packets to their successor component as soon as they receive them. After traversing these *identity* components, packets are silently discarded. This gives us the minimal processing time per packet per component. We also ran a test with no identity component; this acts as a reference: its execution time represents the amount of time spent in reading the trace file, and in the creation and the destruction of packets.

Figure 6 plots the measured overhead (compared to our reference) for each of the traces. This shows approximatively 130 ms<sup>6</sup> per component overhead

<sup>6</sup>We obtained this value by computing the mean overhead by number of layers over each trace, and then performing a linear regression analysis. The squared correlation factor ( $r^2$ ) is equal to 0.9996.

for 500,000 packets (i.e. 260 ns per packet, per component). As one might expect, this value does not depend on the size of packets. This overhead is relatively small, compared to the cost of effective computation time. We can safely say that the layering structure of Pandora has little impact on performance.

### 7.2.2 Demultiplexing

We want to quantify the cost of demultiplexing packets according to their connection identifier. The test consists of extracting TCP packets from traces with various characteristics (presented in Table 3), demultiplexing them, passing them through the identity component and finally discarding them, while timing out (with different timeout values) inactive demultiplexed branches. Our reference is the time taken to extract TCP packets from the traces.

The results are shown in Table 4. It appears that, though the demultiplexing problem is very simple, its cost is rather high. It ranges from 2.1  $\mu$ s to more than 20  $\mu$ s per packet, and has an average around 8  $\mu$ s for the two “realistic” traces: TCP and WWW. First, this confirms that the “layering only” overhead is small (only 0.26  $\mu$ s per packet) and highlights the importance of the algorithm and the data structures used for demultiplexing. Second, it shows that it is very difficult to predict what the cost of demultiplexing will be: it depends on the size of the demultiplexing table and the ratio of packets inserting new entries in the map, and the quantities are very hard to estimate *a priori*.

## 7.3 Web Traffic Collection Overhead

As a final evaluation we use our HTTP reconstruction example. The tasks performed are only a subset of those we previously described: we did not perform IP reassembly (we filtered out fragmented packets), nor the on-line anonymization. As for the previous test, we timeout idle connections (with a timeout value of 255 seconds). To compute the overhead, we take for reference the time needed to reorder TCP packets.

Results are shown in Table 6. It presents the cost of some realistic processing. On average, we require about 50  $\mu$ s per packet. We see then that the layering cost (i.e. splitting into two steps what could be

Name	ANY	TCP	UDP	SYN
Packets	500000	500000	500000	500000
Size (MB)	288.9	268.0	181.2	29.6

Table 2: Characteristics of traces used for layering overhead evaluation.

Trace	SYN			TCP			WWW			WWW-PP		
Total Packets	500000			500000			500000			500000		
Processed Pkts	500000			334771			317572			363107		
Unique Connect. (%)	unknown			11.9			9.8			0.7		
Timeout Value (s)	2	30	255	2	30	255	2	30	255	2	30	255
Map Size (avg)	31	312	2486	780	5788	19736	47	307	1905	19	84	393
Insert/Total (%)	98.7	93.9	93.7	16.9	12.0	11.8	18.5	11.1	10.0	0.4	0.7	0.7

Table 3: Characteristics of traces used for demultiplexing evaluation. The SYN trace is a collection of connection establishment TCP packets. WWW-PP is a trace collected when repeatedly requesting the same document from a Web server with an unique client host. TCP and WWW are “real” traces of TCP and WWW traffic, respectively. The number of processed packets varies since we are discarding acknowledgment-only packets in this experiment. The *Map Size* line is the mean map size in which a new entry was to be inserted. The *Insert/Total* line indicates the ratio between the number of created entries in the map over the total number of processed packets.

Tim.	SYN	TCP	WWW	WWW-PP
2	13.22 (0.14%)	8.21 (0.15%)	7.98 (0.83%)	2.76 (0.65%)
30	18.40 (0.10%)	8.11 (0.20%)	7.54 (0.26%)	2.08 (0.40%)
255	20.3 (0.07%)	7.1 (0.26%)	9.1 (0.32%)	3.8 (0.83%)

Table 4: Evaluation of demultiplexing overhead. Numbers show the average overhead in  $\mu$ s per packet demultiplexed, for the trace and timeout value specified.

Name	WWW1	WWW2	WWW3	WWW-PP
Total Packets	500000	500000	500000	500000
Processed Packets	317572	315992	315826	363107
Unique Connections	31238	24443	21067	2422
Size (MB)	256.6	302.0	272.6	307.5
Requests	21155	21997	18636	119168

Table 5: Characteristics of traces used for HTTP extraction experiments.

Trace	WWW1	WWW2	WWW3	WWW-PP
Overhead	27.8 (0.17%)	63.0 (0.09%)	48.6 (0.11%)	51.1 (0.12%)

Table 6: Average overhead (in  $\mu$ s per packet) for HTTP extraction.

done in one) now represents only 0.52 % of the average time to process a packet. Differences between execution time mainly comes from the different average length of requests.

## 7.4 Web Traffic Collection Throughput

Given that HTTP reconstruction is in fact a real application, we also measure its global throughput. Compared to other tests, this involves an additional step: writing records to a log file. These runs, more than per-packet costs, give us an idea of the maximum bandwidth such an application can monitor without dropping packets. The packets are given to the components at an almost constant rate, which differs greatly from real network conditions.

Table 7 shows the results of these tests. This leads us to think that Pandora can cope with most medium bandwidth links (100 Mb/s), without suffering from too many packet losses. Indeed, except for the WWW-PP trace (where we spent nearly 35 % of our time in writing records), the throughput achieved represents three quarters of the maximum bandwidth. Yet, these results show that Pandora is not fast enough to monitor high bandwidth network, dedicated to HTTP (like ISP backbones).

## 8 Related Work

We borrow the stacking approach from several platforms, in various domains. Well known examples include Ficus [8], *x*-Kernel [9], Horus [17] and Ensemble [7]. To the best of our knowledge, it has never been used for network monitoring tools.

Some intrusion detection systems (IDS) have addressed similar issues. In particular, Network Flight Recorder [16] and Bro [14]. Such tools are used to detect network attacks in real-time, and allow fast response. They both split their processing of incoming packets into distinct stages, each being the responsibility of a specific component. This approach gives such tools the flexibility and the extensibility they require (one cannot know what kind of attack will have to be monitored in the future). They also provide an advanced configuration language to specialize the behavior of the engine. Compared to them, Pandora, uses finer-grained components, and

its configuration relies more on the chaining of specialized components. IDS have a fixed-size stack because packet processing to be performed is well-known and is not likely to change. This allows better performance (we saw Section 7.2 that layering has still a cost), but prevents from using the tool for a different goal.

Among existing monitoring tools, `tcpdump` [11] was the first to be widely used. Its output (a line per packet received) is invaluable for simple network monitoring purpose. For complex cases, the software offers the possibility of dumping a trace of complete packet contents for off-line analysis. Yet, this approach may not be acceptable in situations where the volume of data is too big — on busy, high-speed links, it is not uncommon to collect more than 1 Gigabyte of data within 15 minutes.

Ian Goldberg's IPSE [6] is another recent approach to network monitoring: built as Linux kernel modules, it promises good performance. Yet, such construction does not allow portability and complicates the development of the tool itself. Unfortunately, this software has not been sufficiently documented, preventing us from further investigation.

Windmill [12] is the closest tool to Pandora and looks very similar in spirit to it. Windmill is a platform designed to evaluate protocol performance, via a set of dynamically loaded experiments. Its authors focused on large set of such experiments, each using possibly overlapping components to match their input packets. This lead them to develop their own packet filter (WPF) and to design their protocol modules to avoid redundancy among experiments (e.g. not reassembling twice IP fragments needed by two distinct experiments). Windmill implements protocol extraction through a set of modules statically chained to each other: the inner-most module — i.e. the higher protocol level — recursively calls lower layers, first to let them update their internal state, and then to ask for information they need. For example, the BGP module first lets the TCP module process the packet, and only then asks TCP whether the packet is carrying a FIN flag. In contrast to Pandora, Windmill does not use a "pure" stacking approach and thus perfectly illustrates benefits and drawbacks of both techniques. For example, with Pandora, adding or removing data encryption support, or changing of encryption algorithm is a simple matter of configuration that can be done at run time. Also, implementing an IPv6 module in Windmill can be tricky. On one hand, if we



Trace	WWW1	WWW2	WWW3	WWW-PP
Mb/s	88.3 (0.18%)	75.2 (0.19%)	75.1 (0.12%)	44.8 (0.12%)
reqs./s	868	653	612	2600

Table 7: Throughput of HTTP extraction.

use different interfaces from the IPv4 module, all transport-layer modules have to be updated. On the other hand, if the interface is the same, this prevents from using both IPv4 and IPv6 layers in the same program. However, Windmill should outperform Pandora when executing similar tasks.

With respect to Web traffic monitoring, which motivated the development of Pandora in the first place, several tools have been proposed in the recent past.

HttpFilt and HttpDump [21] were one of the first attempts to construct such specialized tools. No other tool have achieved such complete on-line processing of packets. Unfortunately, their limited performance have lead the authors to give up their development, restricting them to only HTTP/1.0 transactions.

BLT [2, 3] is the more promising one. It is meant to provide on-line HTTP traces, related with lower-layers events (TCP aborted connections, packet loss rate or duplicated, etc.) at a high performance level. With a machine comparable to ours (a 500-MHz Alpha workstation), BLT was able to capture a 12 day trace on an FDDI ring, handling more than 150 millions packets a day (an average of about 1750 packets per second) with a packet loss rate of less than 0.3%. Compared to Pandora, BLT claims better performance. However, unlike Pandora, BLT performs HTTP request/response matching (which is an essential stage in Web logging) off-line.

We regret that none of these recent monitoring tools (namely IPSE, Windmill and BLT) have been publicly released so far, which prevented us from further comparing them with our tool and design choices. This is also partly why we decided to develop Pandora from scratch.

## 9 Conclusion and Future Work

We presented Pandora, a passive network monitoring platform. We discussed its basic components

and how they could be used to set up different monitoring systems at a very low cost. We presented a performance evaluation that supports the claim that Pandora is an efficient flexible continuous monitoring tool, that is: it can run 24 hours a day, 7 days a week. It allows *true* on-line analysis of network protocols, up to the higher levels — including analysis beyond strict application level protocols, such as HTTP matching. The use of components makes it easy to extend the platform.

Nevertheless, Pandora still needs improvements at various levels. First the stack configuration process should benefit from a more user-friendly interface. We also plan to implement a tool that can check stack description validity, in order to avoid runtime errors when feeding a component with packets it does not expect. Last but not least, we must continue developing monitoring components and improving the existing ones.

Pandora has already been used to retrieve HTTP traces during a one month period (representing a total amount of more than six million requests). These traces were then used in Saperlipopette! [15], a tool designed to help dimension proxy cache infrastructure.

More generally, Pandora is meant to facilitate the development of access infrastructure in the ever-growing Internet. It can also be used as a way to compare the various solutions offered to address specific problems. The number and the diversity of tools developed by the research community in the recent past that are related to these issues prove that it corresponds to a real need, and highlight as well the lack of a flexible monitoring tool that could satisfy these.

## Acknowledgments

We owe to thank the anonymous reviewers and our shepherds Vern Paxson and Greg Minshall for their helpful comments. Also thanks to Ian Piumarta and

Thomas Colcombet for their useful suggestions from the early beginning of this work.

## Availability

The source code is publicly available under the GPL license in a beta, undocumented version. Documentation is planned to be written in the near future. See <http://www-sor.inria.fr/projects/relais/pandora/> for release information.

## References

- [1] Ramon Caceres, Cormac J. Sreenan, and J. E. van der Merwe. mmdump - a tool for monitoring multimedia usage on the internet. Technical Report TR 00.2.1, AT&T Labs-Research, February 2000. <http://www.research.att.com/~ramon/papers/mmdump.ps.gz>.
- [2] Anja Feldmann. Continuous online extraction of HTTP traces from packet traces. Position paper for the W3C Web Characterization Group Workshop, November 1998. [http://www.research.att.com/~anja/feldmann/w3c98\\_httptrace.abs.html](http://www.research.att.com/~anja/feldmann/w3c98_httptrace.abs.html).
- [3] Anja Feldmann, Ramon Caceres, Fred Douglass, Gideon Glass, and Michael Rabinovich. Performance of Web proxy caching in heterogeneous bandwidth environments. In *Proceedings of the INFOCOM '99 conference*, March 1999. [http://www.research.att.com/~anja/feldmann/papers/infocom99\\_proxim.ps.gz](http://www.research.att.com/~anja/feldmann/papers/infocom99_proxim.ps.gz).
- [4] Anja Feldmann, Jennifer Rexford, and Ramon Caceres. Efficient policies for carrying web traffic over flow-switched networks. *IEEE/ACM Transactions*, December 1998. [http://www.research.att.com/~anja/feldmann/papers/ton98\\_flow.ps.gz](http://www.research.att.com/~anja/feldmann/papers/ton98_flow.ps.gz).
- [5] Robert Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, and Tim Berners-Lee. Hypertext Transfer Protocol - HTTP/1.1. Request for Comments 2068, January 1997. [ftp://ftp.isi.edu/in-notes/rfc2068.txt](http://ftp.isi.edu/in-notes/rfc2068.txt).
- [6] Steven D. Gribble and Eric A. Brewer. System design issues for Internet middleware services: Deductions from a large client trace. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, Monterey, CA, December 1997. [http://www.cs.berkeley.edu/~gribble/papers/sys\\_trace.ps.gz](http://www.cs.berkeley.edu/~gribble/papers/sys_trace.ps.gz).
- [7] Mark Hayden. The ensemble system. Technical Report TR98-1662, Cornell University, January 1998. <http://cs-tr.cs.cornell.edu/Dienst/UI/1.0/Display/ncstrl.cornell/TR98-1662>.
- [8] John S. Heidemann. Stackable layers: An architecture for file system development. Technical Report UCLA-CSD 910056, University of California, Los Angeles, CA (USA), July 1991. <http://www.isi.edu/~johnh/PAPERS/Heidemann91c.ps.gz>.
- [9] Norman C. Hutchinson and Larry L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64-76, January 1991. [ftp://ftp.cs.arizona.edu/xkernel/Papers/architecture.ps](http://ftp.cs.arizona.edu/xkernel/Papers/architecture.ps).
- [10] Van Jacobson, Craig Leres, and Steven McCane. libpcap. software (latest release: version 0.4), June 1989. [ftp://ftp.ee.lbl.gov/libpcap.tar.Z](http://ftp.ee.lbl.gov/libpcap.tar.Z).
- [11] Van Jacobson, Craig Leres, and Steven McCane. tcpdump. software (latest release: version 3.4), June 1989. [ftp://ftp.ee.lbl.gov/tcpdump.tar.Z](http://ftp.ee.lbl.gov/tcpdump.tar.Z).
- [12] G. Robert Malan and Farnam Jahanian. An extensible probe architecture for network protocol performance measurement. In *Proceedings of ACM SIGCOMM '98*, Vancouver, British Columbia, September 1998. <http://www.eecs.umich.edu/~rmalan/publications/mjSigcomm98.ps.gz>.
- [13] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX conference*, San Diego, California, January 1993. <http://www.ntua.gr/rin/docs/bpf-usenix93.ps>.
- [14] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998. [ftp://ftp.ee.lbl.gov/papers/bro-usenix98-revised.ps.Z](http://ftp.ee.lbl.gov/papers/bro-usenix98-revised.ps.Z).
- [15] Guillaume Pierre and Mésaac Makpangou. Saperlipopette!: a distributed Web caching systems evaluation tool. In *Proceedings of the 1998 Middleware conference*, pages 389-405, September 1998. [http://www-sor.inria.fr/publi/SDWCSET\\_middleware98.html](http://www-sor.inria.fr/publi/SDWCSET_middleware98.html).
- [16] Marcus J. Ranum, Kent Landfield, Mike Stolarchuk, Mark Sienkiewicz, Andrew Lambeth, and Eric Wall. Implementing a generalized tool for network monitoring. In *Proceedings of the Eleventh Systems Administration Conference (LISA '97)*, San Diego, California, October 1997. [http://www.usenix.org/publications/library/proceedings/lisa97/full\\_papers/rs/01.ranum/01.pdf](http://www.usenix.org/publications/library/proceedings/lisa97/full_papers/rs/01.ranum/01.pdf).
- [17] Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr. A framework for protocol composition in horus. In *Proceedings of Principles of Distributed Computing*, August 1995. <http://www.cs.cornell.edu/Info/People/rvr/papers/podc/podc.html>.
- [18] Duane Wessels. The Squid Internet object cache. National Laboratory for Applied Network Research/UCSD, software, 1997. <http://www.squid-cache.org/>.
- [19] Duane Wessels and K. Claffy. Internet Cache Protocol (ICP), version 2. National Laboratory for Applied Network Research/UCSD, Request for Comments 2186, September 1997. [ftp://ftp.isi.edu/in-notes/rfc2186.txt](http://ftp.isi.edu/in-notes/rfc2186.txt).
- [20] Duane Wessels and K. Claffy. ICP and the Squid web cache. *IEEE Journal on Selected Areas in Communication*, 16(3):345-357, April 1998. <http://www.ircache.net/~wessels/Papers/icp-squid.ps.gz>.
- [21] Roland Wooster, Stephen Williams, and Patrick Brooks. Httpdump: Network HTTP packet snooper. working paper, April 1996. [http://ei.cs.vt.edu/~succeed/96httpdump/final\\_paper/paper.ps.gz](http://ei.cs.vt.edu/~succeed/96httpdump/final_paper/paper.ps.gz).



# A Comparison of File System Workloads

Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson  
University of California, Berkeley and University of Washington  
{drew, lorch}@cs.berkeley.edu, tom@cs.washington.edu

## Abstract

*In this paper, we describe the collection and analysis of file system traces from a variety of different environments, including both UNIX and NT systems, clients and servers, and instructional and production systems. Our goal is to understand how modern workloads affect the ability of file systems to provide high performance to users. Because of the increasing gap between processor speed and disk latency, file system performance is largely determined by its disk behavior. Therefore we primarily focus on the disk I/O aspects of the traces. We find that more processes access files via the memory-map interface than through the read interface. However, because many processes memory-map a small set of files, these files are likely to be cached. We also find that file access has a bimodal distribution pattern: some files are written repeatedly without being read; other files are almost exclusively read. We develop a new metric for measuring file lifetime that accounts for files that are never deleted. Using this metric, we find that the average block lifetime for some workloads is significantly longer than the 30-second write delay used by many file systems. However, all workloads show lifetime locality: the same files tend to be overwritten multiple times.*

## 1 Introduction

Like other computer systems, file systems provide good performance by optimizing for common usage patterns. Unfortunately, usage patterns vary both over time and across different user communities. To help delineate current workload patterns, we decided to measure a wide range of file systems in a number of different environments, specifically, UNIX and Windows NT, client and server, instructional, research, and production. We compare our results with those from the Sprite study, conducted in 1991. Although we were interested in tracking how behavior has changed since the Sprite study, we do not directly reproduce all of their results. Their study

focused on cache and virtual memory behavior. Since the relative performance of hardware has changed since that time, we focus instead on the I/O bottleneck.

We collected traces from four different groups of machines. Three of the groups run HP-UX, a variant of the UNIX operating system. One of these is an instructional laboratory, another is a set of computers used for research, and another is a single web server. The last group is a set of personal computers running Windows NT. This diversity of traces allows us to make conclusions not only on how current file system usage differs from past file system usage, but also how file system usage varies among machines used for different purposes.

Because improvements in disk latency are increasingly lagging behind those of processors and disk bandwidth, we chose to focus our study on measurements that elucidate how disk behavior is affected by workload and file system parameters. As the I/O gap grows, one way to provide good performance is to cache as many file reads and writes as possible and to minimize latencies for the remainder. For example, one way to avoid disk reads is by employing large file caches. Our results show that while small caches can avert many disk reads, there are diminishing benefits for large cache sizes. In addition to file reads, memory-mapping has become a popular file access method. We examine memory-mapping behavior in order to see the effect of memory-mapped files on the file cache. We find that more processes access files via memory-mapping than through reads or writes. For the UNIX workloads, we find that a small set of memory-mapped files tend to be shared among many processes. As a result, cache misses on these files are unlikely.

To avoid disk writes, the file system can increase the time between an application's write and flushing the data to disk, for example, by using NVRAM. By delaying writes, blocks that are deleted in the interval need not be written at all. We find that most blocks live longer than the standard 30-second write delay commonly employed by file systems. In UNIX systems, most blocks die within an hour; in NT, many blocks survive over a day. Most blocks die due to overwrites, and these overwrites have a high degree of locality—that is, most overwritten files are multiply overwritten. Because of this locality, even a small write buffer is sufficient to handle a day's

---

This research was supported by the National Science Foundation (Grant No. CCR-9972244), the State of California MICRO program, Cisco Systems, Fujitsu Microelectronics, IBM, Intel Corporation, Maxtor Corporation, Microsoft Corporation, Quantum Corporation, Sony Research Laboratories, Sun Microsystems, Toshiba Corporation, and Veritas Software. In addition, Roselli was supported by a GAANN fellowship.

worth of write traffic.

To reduce disk seeks, most file systems organize their layout to optimize for either reads or writes. We find that whether read traffic or write traffic dominates varies depending on the workload and the file system configuration. However, for all workloads, we find that individual files tend to have bimodal access patterns—they are either read-mostly or write-mostly. This tendency is most clear in frequently accessed files.

## 2 Related Work

Characterizing file system behavior is difficult due to both the wide range of workloads and the difficulty in obtaining data to analyze. Obviously, no trace analysis project has the scope to analyze all relevant features of all relevant workloads. Instead, each study lets us understand a piece of the greater picture.

In order to minimize the complexity of trace collection, many studies concentrate on static data, which they collect by examining file system metadata at one or several frozen instants in time [Douc99] [Sien94] [Chia93] [Benn91] [Saty81] [Smit81]. These studies of *snapshots* are useful for studying distributions of file attributes commonly stored in metadata, such as file size, last access time, last modification time, file name, and directory structure.

Dynamic traces of continuous file system access patterns yield more detailed information about file system usage. However, these traces are considerably more difficult to collect both because of the volume of data involved and because the collection process typically involves modifying the operating system kernel. Some tracing methods avoid altering the kernel by recording file system events that pass over a network [Blaz92] [Dahl94]. However, this method misses file system events that do not cross the network, such as local file system calls. Also, artifacts of the network file system being measured can affect these types of traces.

Modifying the kernel to obtain local file system behavior has its own set of drawbacks. First, the kernel source code is not always available. Second, the modified kernels must be deployed to users willing to run their applications on an altered kernel. Finally, the overhead of collecting fine-grained traces must be kept low so that overall system performance is not significantly degraded. Due to these limitations, most researchers limit their trace collection to only the data that is necessary to perform specific studies. For example, the traces collected to perform analysis of directory access behav-

ior in [Floy89] do not include file read or write requests. The disk activity study in [Ruem93] is at the disk level and does not include specific file system calls. Mummert et al. focused on results relevant to disconnected file system operation [Mumm94]. Zhou and Smith collected traces on personal computers for research in low-power computing [Zhou99].

In 1985, Ousterhout et al. presented a general characterization of dynamically collected traces [Oust85]. In this work, they traced three servers running BSD UNIX for slightly over three days. This paper introduced a framework for workload analysis using metrics such as run length, burstiness, lifetime of newly written bytes, and file access sequentiality. Henceforth, we refer to this work as the BSD study. In 1991, Baker et al. conducted the same type of analysis on four two-day sets of traces of the Sprite file system [Bake91]. They collected these traces at the file servers and augmented them with client information on local cache activity. For the rest of this paper, we refer to this work as the Sprite study. The data analysis techniques developed in the BSD and Sprite studies were repeated in several subsequent studies. In 1991, Bozman et al. repeated many of the Sprite studies using traces from two separate IBM sites [Bozm91]. This study confirmed that the results from the Sprite study applied to non-academic sites. In 1999, the same studies were repeated on three sets of two-week traces taken from 45 hosts running Windows NT [Voge99]. This workload is close to our NT workload, and for the analyses that are directly comparable (file size, file lifetime and access patterns), our results are similar.

In this work, we repeat some of the influential studies introduced by the BSD study, such as file access patterns. In addition, we contribute new studies that have become relevant to modern systems, such as the effect of memory-mapping files on the file cache. A more complete comparison of the Sprite studies against our UNIX traces can be found elsewhere [Rose98]. Because the Sprite traces are publicly available, we generate results for the Sprite traces wherever possible for purposes of comparison.

## 3 Trace Collection

### 3.1 Environment

We collected the traces discussed in this paper in four separate environments. Three of these environments use Hewlett-Packard series 700 workstations running HP-UX 9.05. Each of the HP-UX machines has 64MB of memory. The first group consists of twenty machines

located in laboratories for undergraduate classes. For the rest of this paper, we refer to this workload as the Instructional Workload (INS). The second group consists of 13 machines on the desktops of graduate students, faculty, and administrative staff of our research group project. We refer to this workload as the Research Workload (RES). Of all our traces, the environment for this workload most closely resembles the environment in which the Sprite traces were collected. We collected the third set of traces from a single machine that is the web server for an online library project. This host maintains a database of images using the Postgres database management system and exports the images via its web interface. This server received approximately 2,300 accesses per day during the period of the trace. We refer to this as the WEB workload. The INS machines mount home directories and common binaries from a non-traced Hewlett-Packard workstation. In total, we collected eight months of traces from the INS cluster (two semesters), one year of traces from the RES cluster, and approximately one month of traces from the WEB host.

We collected the fourth group of traces from eight desktop machines running Windows NT 4.0. Two of these machines are 450 MHz Pentium IIIs, two are 200 MHz Pentium Pros, and the other four are Pentium IIs ranging from 266–400 MHz. Five of them have 128 MB of main memory, while the others have 64, 96, and 256 MB. These hosts are used for a variety of purposes. Two are used by a crime laboratory director and his supervisor, a state police captain; they use these machines for time management, personnel management, accounting, procurement, mail, office suite applications, and web browsing and publishing. Another two are used for networking and system administration tasks: one primarily runs an X server, email client, web browser, and Windows NT system administration tools; the other primarily runs office suite, groupware, firewall, and web browsing applications. Two are used by computer science graduate students as X servers as well as for software development, mail, and web browsing. Another is shared among the members of a computer science graduate research group and used primarily for office suite applications. The final machine is used primarily as an X server, but occasionally for office suite and web browsing applications. Despite the different uses of the NT machines, the results are similar for all the machines, so we include them together as one group.

## 3.2 Trace Collection Methodology

We used separate tools to collect traces for the HP-UX and Windows NT systems. While both of our collection techniques trace similar file system events, their imple-

mentations are quite different.

### 3.2.1 HP-UX Collection Methodology

For the UNIX machines, we used the auditing subsystem to record file system events. Although the auditing system was designed for security purposes, it is ideal for tracing since it catches the logical level of requests using already-existing kernel functionality. The auditing subsystem gets invoked after a system call and is configured to log specified system calls with their arguments and return values. However, it does not record kernel file system activity, such as paging from executable images.

The major problem we faced in using the auditing system was that HP-UX records pathnames exactly as specified by the user, and users often specify paths relative to the current working directory instead of with their complete paths. Since some file systems use a file's parent directory to direct file layout, we needed to record the full pathname. We solved this problem by recording the current working directory's pathname for each process and configuring the auditing system to catch all system calls capable of changing the current working directory. These changes required only small changes to the kernel (about 350 of lines of C code) and were wholly contained within the auditing subsystem.

### 3.2.2 Windows NT Collection Methodology

We collected the Windows NT traces using a tool we developed that traces not only file system activity, but also a wide range of device and process behavior [Lorc00]. We focus here on the aspects of the tracer relevant to tracing file system activity.

We perform most of the file system tracing using the standard mechanism in Windows NT for interposing file system calls: a file system filter driver. A file system filter driver creates a virtual file system device that intercepts all requests to an existing file system device and handles them itself. Our filter device merely records information about the request, passes the request on to the real file system, and arranges to be called again when the request has completed so it can record information about the success or failure of the request. The design of our filter driver borrows much from the Filemon file system monitoring program [Russ97b].

A Windows NT optimization called the *fast path* complicates tracing these file systems. The operating system uses this optimization whenever it believes a request can be handled quickly, for example, with the cache. In this



case, it makes a call to a fast-dispatch function provided by the file system instead of passing requests through the standard request path. In order to intercept these calls, we implemented our own fast-dispatch functions to record any calls made this way.

In order to collect data on memory-mapping operations, we needed to interpose Windows NT system calls. This is difficult because Microsoft gives no documented way to do this. Fortunately, a tool called Regmon solves this problem; it finds the system call entry point vector in memory and overwrites certain entry points with our own [Russ97a].

Because we interpose at the file system layer and not at the system call layer, there were some challenges in converting our traces to a format comparable with the UNIX traces. The first problem arises when the file system calls the cache manager to handle a read request, and there is a miss. The cache manager fills the needed cache block by recursively calling the file system. We need to identify the recursive requests because they do not reflect actual read requests and should be elided. We distinguish them by three of their properties: they are initiated by the kernel, they have the no-caching flag set (in order to prevent an infinite loop), and they involve bytes that are being read by another ongoing request. The second problem is that we cannot distinguish a read caused by an explicit read request from one caused by kernel-initiated read-ahead. We distinguish the latter by looking for read requests with the following four properties: they are initiated by the kernel, they have the no-caching flag set, they do not involve bytes currently being read by another request, and they are made to a file handle that was explicitly read earlier. Finally, it is also difficult to determine which read and write requests are due to paging of memory-mapped files. If a request is initiated by the kernel with the no-caching flag set and it does not belong to any of the previous characterizations, we classify it as a paging request.

The file system interface of Windows NT is quite different from that of UNIX. For instance, there is no `stat` system call in Windows NT, but there is a similar system call: `ZwQueryAttributesFile`. For the purpose of comparison, we have mapped the request types seen in Windows NT to their closest analogous system calls in UNIX in this paper.

## 4 Results

Due to the time-consuming nature of collecting statistics on the entire length of our traces (which are currently over 150GB compressed), we present results in this

paper based on subsets of the traces. For the INS and RES traces, we used traces collected from the month of March 1997. For WEB, we used the traces from January 23 to February 16, 1997. Because this trace includes activity not related to the web server, we filtered it to remove non-web-server activity. Because the NT traces begin at different times, we chose a 31-day period for each host. All but one of these periods were within the first quarter of the year 2000; the other trace was taken from October and November of 1999. For the Sprite results, our results differ slightly from those presented by Hartman and Ousterhout [Hart93] because we filter them differently. For example, we do not include non-file, non-directory objects in any results.

None of our results include paging of executables. For the NT workload, executable paging constitutes 15% of all reads and nearly 30% of all writes. Paging activity for the UNIX workloads is unknown.

### 4.1 Histogram of Key Calls

To provide an overview of our workloads, we first present counts of the most common events traced; these are summarized in Table 1. The results reveal some notable differences among the workloads. For example, the WEB workload reads significantly more data than the other workloads; its read to write ratio is two orders of magnitude higher than any other workload. The NT workload reads and writes more than twice the amount of data per host per day than the INS and RES workloads, despite having significantly fewer users. Also, notable in all workloads is the high number of requests to read file attributes. In particular, calls to `stat` (including `fstat`) comprise 42% of all file-system-related calls in INS, 71% for RES, 10% for WEB, and 26% for NT.

Two common usage patterns could account for the large number of `stat` calls. First, listing a directory often involves checking the attributes of each file in the directory: a `stat` system call is made for each file. Second, a program may call `stat` to check attributes before opening and accessing a file. For example, the `make` program checks the last modification times on source and object files to determine whether to regenerate the object file. We measured the percentage of `stat` calls that follow another `stat` system call to a file from the same directory to be 98% for INS and RES, 67% for WEB, and 97% for NT. The percentage of `stat` calls that are followed within five minutes by an open to the same file is 23% for INS, 3% for RES, 38% for WEB, and only 0.7% for NT.



TABLE 1. Trace Event Summary

	INS	RES	WEB	NT	Sprite
hosts	19	13	1	8	55
users	326	50	7	8	76
days	31	31	24	31	8
data read (MB)	94619	52743	327838	125323	42929
data written (MB)	16804	14105	960	19802	9295
read:write ratio	5.6	3.7	341.5	6.3	4.6
all events (thousands)	317859	112260	112260	145043	4602
fork (thousands)	4275	1742	196	NA	NA
exec (thousands)	2020	779	319	NA	NA
exit (thousands)	2107	867	328	NA	NA
open (thousands)	39879	4972	6459	21583	1190
close (thousands)	40511	5582	6470	21785	1147
read (thousands)	71869	9433	9545	39280	1662
write (thousands)	4650	2216	779	7163	455
mem. map (thousands)	7511	2876	1856	614	NA
stat (thousands)	135886	79839	3078	37035	NA
get attr. (thousands)	1175	826	15	36	NA
set attr. (thousands)	467	160	23	273	NA
chdir (thousands)	1262	348	80	NA	NA
read dir. (thousands)	4009	1631	172	12486	NA
unlink (thousands)	490	182	2	285	106
truncate (thousands)	37	4	0	1981	42
fsync (thousands)	514	420	2	1533	NA
sync (thousands)	3	71	0	NA	NA

This table summarizes the number of events for the time period indicated for each trace. For all workloads, the above calls represent over 99% of all traced calls. The get attribute category includes `getacl`, `fgetacl`, `access`, and `getaccess`. The set attribute category includes `chmod`, `chown`, `utime`, `fchmod`, `fchown`, `setacl`, and `fsetacl`. The number of users is estimated from the number of unique user identifiers seen. This may be an overestimate since some user identifiers are simply administrative. For the NT traces, `exec` and `chdir` calls were not recorded, and process forks and exits were recorded only periodically during the NT traces.

Since this system call is so common, it would be worthwhile to optimize its performance. Since it is most commonly invoked near other `stat` calls in the same directory, storing the attribute data structures together with those from the same directory [McKu84] or within the directory structure [Gang97] may provide better performance than storing each file's attribute information with its data blocks.

## 4.2 Data Lifetime

In this section, we examine block lifetime, which we define to be the time between a block's creation and its deletion. Knowing the average block lifetime for a work-

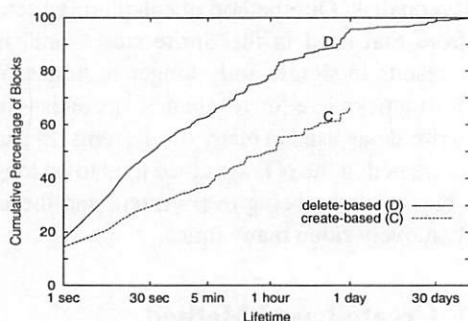
load is important in determining appropriate write delay times and in deciding how long to wait before reorganizing data on disk. Our method of calculating lifetime differs from that used in the Sprite study, and, in some cases, results in significantly longer lifetimes. We find that most blocks live longer than 30 seconds—the standard write-delay used in many file systems. In particular, blocks created in the NT workload tend to be long-lived. Most blocks die by being overwritten, and these blocks are often overwritten many times.

### 4.2.1 Create-based Method

We calculate lifetime by subtracting a block's creation time from its deletion time. This is different from the *delete-based* method used by [Bake91] in which they track all deleted files and calculate lifetime by subtracting the file's creation time from its deletion time. In our *create-based* method, a trace is divided into two parts. We collect information about blocks created within the first part of the trace. We call the second part of the trace the *end margin*. If a tracked block is deleted during either part of the trace, we calculate its lifetime by subtracting the creation time from the deletion time. If a tracked block is not deleted during the trace, we know the block has lived for at least the end margin.

The main difference between the create-based and delete-based methods is the set of blocks that we use to generate the results. Because the delete-based method bases its data on blocks that are deleted, one cannot generalize from this data the lifetime distribution of newly created blocks. Because that is the quantity which interests us, we use the create-based algorithm for all results in this paper. One drawback of this approach is that it only provides accurate lifetime distributions for lifetimes less than the end margin, which is necessarily less than the trace duration. However, since our traces are long-term, we are able to acquire lifetime data sufficient for our purposes; we use an end margin of one day for all results in this section. Figure 1 shows the difference in results of create-based and delete-based methods on one of the Sprite traces. Due to the difference in sampled files, the delete-based method calculates a shorter lifetime than the create-based method.

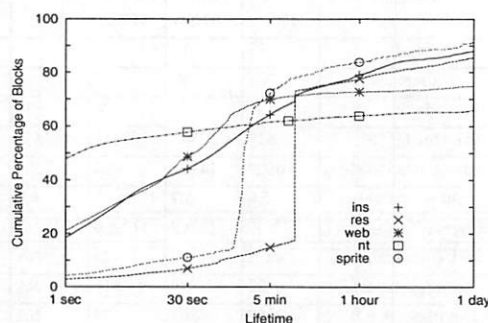
If the traces collected reflect random samples of the steady state of creation and deletion, the principal difference between the methods would result from blocks that are created and never deleted. As a result of this difference, the create-based method predicts that disk space used will tend to increase with time—something disk sales confirm.



**FIGURE 1. Create-based versus Delete-based Lifetime Distributions.** This graph shows byte lifetime values calculated using a create-based and a delete-based algorithm. The trace used comprises the two contiguous days represented in the fourth Sprite trace (days 7 and 8); this trace showed the most difference between the two methods of all the Sprite traces. Unlike the results reported in [Bake91], these results include blocks overwritten in files that were not deleted, however this difference has only minor effects on the results.

## 4.2.2 Block Lifetime

Using the create-based metric for both our traces and the Sprite traces, we calculate block lifetimes using a block size of 512 bytes. Figure 2 shows these results. Block lifetime for a combination of the Sprite traces is included for comparison. Because most activity occurred during the second trace, this trace dominates Sprite's lifetime results. The graph shows a knee in the WEB workload that is mainly due to database working space files and http log files. RES has a knee at ten minutes caused primarily by periodic updates to Netscape database files. The Sprite trace has a knee just before five minutes contributed mainly by activity in the second trace. Since the Sprite traces do not include information on filenames, we do not know which files were deleted at that time. Neither INS nor NT has a knee; instead, block lifetimes gradually decrease after one second. Unlike the other workloads, NT shows a bimodal distribution pattern—nearly all blocks either die within a second or live longer than a day. Although only 30% of NT block writes die within a day, 86% of newly created files die within that timespan, so many of the long-lived blocks belong to large files. Some of the largest files resulted from newly installed software. Others were in temporary directories or in the recycle bins on hosts where the bin is not emptied immediately. Of the short-lived blocks, many belong to browser cache and database files, system registry and log files, and files in the recycle bin on hosts where users immediately empty the bin.

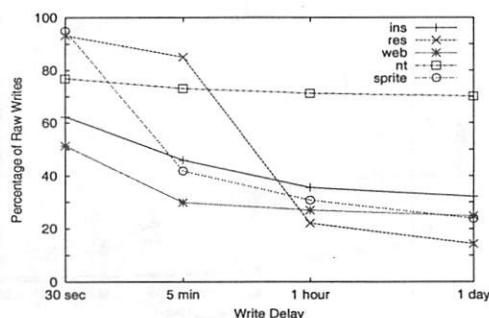


**FIGURE 2. Block Lifetime.** This graph shows create-based block lifetimes using a block size of 512 bytes. Points demarcate the 30 second, 5 minute, and 1 hour points in each curve. The end margin is set to 1 day for these results.

## 4.2.3 Lifetime Locality

By recording whether blocks die due to file deletion, truncation, or overwriting, we observe that most blocks die due to overwrites. For INS, 51% of blocks that are created and killed within the trace die due to overwriting; for RES, 91% are overwritten; for WEB, 97% are overwritten; for NT, 86% are overwritten. A closer examination of the data shows a high degree of locality in overwritten files. For INS, 3% of all files created during the trace are responsible for all overwrites. These files are overwritten an average of 15 times each. For RES, 2% of created files are overwritten, with each file overwritten an average of 160 times. For WEB, 5% of created files are overwritten, and the average number of overwrites for these files is over 6,300. For NT, 2% of created files are overwritten; these files are overwritten an average of 251 times each. In general, a relatively small set of files are repeatedly overwritten, causing many of the new writes and deletions.

An important result from this section is that average block lifetime is longer than delete-based lifetime estimates would predict. For some workloads, average block lifetime is significantly longer than the standard file system write delay of 30 seconds. Since it is unreasonable to leave data volatile for a longer period of time, file system designers will need to explore alternatives that will support fast writes for short-lived data. Some possibilities are NVRAM [Bake92] [Hitz94], reliable memory systems [Chen96], backing up data to the memory of another host, or logging data to disk. Most file blocks die in overwrites, and the locality of overwrites offers some predictability that may prove useful to the file system in determining its storage strategy.



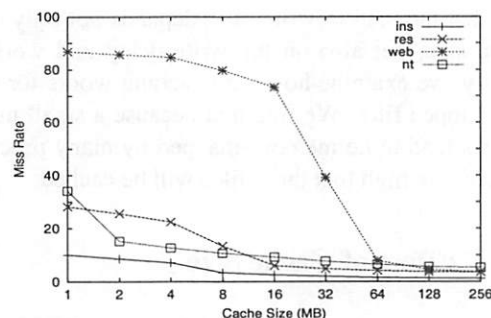
**FIGURE 3. Write Bandwidth versus Write Delay.** Using a simulated 16MB write buffer and varied write delay, we show the percentage of all writes that would be written to disk. For these results, we ignore calls to `sync` and `fsync`.

### 4.3 Effect of Write Delay

Since newly written blocks often live longer than thirty seconds, increasing the write delay period should reduce disk write traffic. However, two factors limit the effectiveness of increasing write delay. First, user requests to `sync` and `fsync` cause data to be written to disk whether or not the write delay period has passed. Second, the operating system may limit the amount of dirty data that may be cached. This limit is generally imposed so that reading a new page into the cache is not slowed by the need to write out the old page first. On systems with NVRAM, the size limit is simply imposed by the NVRAM capacity. In either case, we refer to the space allocated to dirty pages as the write buffer.

In order to measure the effectiveness of increasing write delay, we simulated a write buffer and measured the resultant disk bandwidth while varying the write delay and the capacity of the buffer. Figure 3 shows the results using a 16MB buffer. For these results, we ignore calls to `sync` and `fsync`. As expected, the efficacy of increasing write delay is strongly related to the average block lifetime for each workload. Since RES has many blocks that live less than one hour, a one-hour write delay significantly throttles disk write traffic. On the other hand, the NT workload contains more long-lived blocks, so even write delays of a day have little effect.

To estimate the memory capacity needed to increase write delay, we tested write buffers of size 4MB and 16MB, and an infinitely-sized write buffer. For all workloads, the 16MB buffer closely approximates an infinitely-sized write buffer. In fact, for all workloads except Sprite, the 4MB write buffer also approximates an infinitely-sized write buffer. Large simulations included in the second Sprite trace (the third and fourth of the eight days) are probably responsible for the large write band-



**FIGURE 4. Read Bandwidth versus Cache Size.** This graph shows the percentage of all block read requests that miss the cache versus cache size. The block size used by the cache simulator is 4KB. The cache was warmed with a day of traces before generating results.

width. When these traces are omitted, the 4MB write buffer approximates an infinitely-sized buffer for the Sprite workload as well.

The importance of user calls to `sync` and `fsync` to flush data to reliable storage depends on the storage strategy employed. For example, a file system using NVRAM may ignore these calls since the data is already reliably stored. On other systems, the longer the data is kept in the write buffer, the stronger the impact of these calls. In our study, the maximal impact would be to the infinitely-sized write buffer with a write delay period of one day. For INS, calls to flush data increased writes to disk by 8% at this point; for RES, these calls increased write bandwidth by 6%. For NT, write bandwidth increased by 9%, and for WEB there was no increase at all.

In summary, the efficacy of increasing write delay depends on the average block lifetime of the workload. For nearly all workloads, a small write buffer is sufficient even for write delays of up to a day. User calls to flush data to disk have little effect on any workload.

### 4.4 Cache Efficacy

An important factor in file system performance is how effectively the cache absorbs read requests. In particular, we are interested in how effective caches are at reducing disk seeks and how caching affects the balance between disk reads and writes. In this section, we examine the effect of cache size on read misses. We find that even relatively small caches absorb most read traffic, but there are diminishing returns to using larger caches. We also examine how caching affects the ratio of disk reads to disk writes. In 1992, Rosenblum and Ousterhout claimed that large caches would avert most disk reads, so file system layout should optimize for disk writes [Rose92]. We



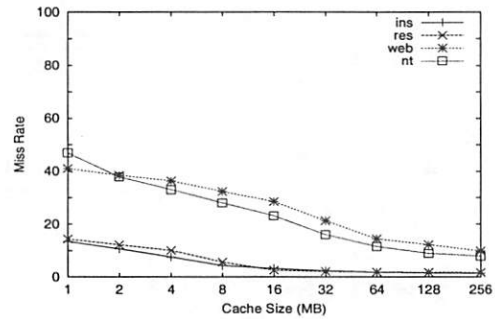
find that the read to write ratio depends not only on the cache size, but also on the write delay and workload. Finally, we examine how well caching works for memory-mapped files. We find that because a small number of files tend to be memory-mapped by many processes, chances are high that these files will be cached.

#### 4.4.1 Effect of Cache Size

We implemented a cache simulator to test the effectiveness of different cache sizes on read traffic. Both reads and writes enter blocks into the simulator, and blocks are replaced in LRU order. For all results in this section, we modeled a local cache, so each host maintains its own instance of the simulator.

Figure 4 shows the cache miss bandwidth for reads for various cache sizes. For all workloads, the curves have a knee showing the working set size, and there are diminishing benefits to increasing the cache size beyond this point. The WEB workload has the largest working set size; its read bandwidth does not reach the point of diminishing returns until a cache size of 64MB. Some of its poor performance may be due to the LRU replacement policy interacting poorly with the database engine. For the other workloads, even a 1MB cache reduces read bandwidth by 65–90%. For these workloads, there is little benefit to increasing the cache beyond 16MB. The BSD study predicted that in the future larger caches would significantly reduce disk reads. However, several years later, the Sprite study found that despite its large caches, read misses did not decrease as much as expected. Our results show that even very large caches have limited effectiveness in reducing read misses.

Since disk bandwidth is improving faster than disk latency, a critical metric in evaluating cache performance is the number of seeks caused by cache misses. Most file systems attempt to store blocks from the same file consecutively on disk. For example, FFS specifically allocates new file blocks as closely as possible to previous file blocks [McVo91]. In LFS, blocks are laid out in the order they are written [Rose92]. Since most files are written sequentially (as we show in Section 4.6), file blocks tend to be allocated consecutively on disk. If file blocks are laid out on disk consecutively, a rough estimate for the number of seeks incurred is a count of the disk reads to different files. We call this metric *file read misses* and calculate it as follows. Within a stream of cache misses, if a cache miss is to the same file as the previous cache miss, we count no file read miss; otherwise, we increment the number of file read misses by one. We define the *file write miss* metric analogously. Although these are crude metrics, we believe they are more accurate esti-



**FIGURE 5. File Reads versus Cache Size.** The miss rate is the percentage of file read misses out of the raw number of file reads. This graph shows the file miss rate for various cache sizes. The block size used by the cache simulator is 4KB. The cache was warmed with a day of traces before results were collected.

mates of seeks than block miss counts.

When multiple hosts share a single file system, a strict computation of the file read count requires interleaving the traces for those hosts. Because the INS and RES clusters share file servers for most of their file system activity, we were able to estimate the effect of file server sharing on file reads by running our measurements on these workloads using both a single interleaved trace for all hosts together and separate traces for each host. These two methods show at most a 2% difference in file read counts and no difference at all when the cache size is over 16MB. This may be because file system traffic tends to be bursty[Grib98]—bursts of activity from single streams may cause a series of cache misses near enough to each other in time that there are few intervening cache misses from other processes in the same time period.

In Figure 5, we show the effectiveness of different cache sizes on reducing the number of file read misses, using interleaved traces when applicable. The graph shows that even a 1MB cache is sufficient to more than halve the number of file read misses for all workloads. At the 1MB cache size, the WEB workload has many fewer file read misses than block read misses, which indicates that many block misses are part of larger files.

#### 4.4.2 Read and Write Traffic

File systems lay out data on disk to optimize for reads [McKu84] or writes [Rose92] [Hitz94], depending on which type of traffic is likely to dominate. As we have already shown, the amount of disk write traffic depends largely on the write delay and the amount of read traffic depends on the cache size. In order to compare the amount of read and write traffic, we examine two envi-



TABLE 2. I/O Count

	INS	RES	WEB	NT
<b>Impoverished Environment</b>				
Block Reads	4,417,055	1,943,728	70,658,318	2,820,438
Block Writes	909,120	2,970,596	1,646,023	3,420,874
File Reads	620,752	199,436	2,389,988	330,528
File Writes	524,551	247,960	144,155	341,581
<b>Enriched Environment</b>				
Block Reads	2,114,991	613,077	6,544,037	1,761,339
Block Writes	1,510,163	585,768	1,483,862	3,155,584
File Reads	277,155	70,078	980,918	144,575
File Writes	209,113	101,621	64,246	248,883

In the impoverished environment, read results are based on an 8MB local cache and write results are based on a 16MB write buffer with a 30 second write delay. In the enriched environment, read results are based on a 64MB local cache, and write results are based on a 16MB write buffer with a 1 hour delay. In both environments, the block size is 4KB, and calls to `sync` and `fsync` flush the appropriate blocks to disk whether or not the write delay has elapsed.

ronments. The first environment has 8MB of local cache and a write delay of 30 seconds; we refer to this as the impoverished environment. The second, the enriched environment, has 64MB of local cache and a write delay of 1 hour. Read and write traffic for each environment are shown in Table 2. By looking at the number of blocks read and written, we see that reads dominate writes in all cases for the WEB workload. For the INS workload, the number of read blocks is almost five times the number of write blocks in the impoverished environment but is only about 50% greater in the enriched environment. For the RES workload, writes dominate the impoverished environment. In the enriched environment, there are more block reads than block writes but fewer file reads than writes. This is most likely caused by the large number of small writes made to various log files on the RES hosts. For the NT workload, writes dominate reads in all cases. However, most of the write traffic is caused by a single host. When this host is removed, reads dominate writes for all categories except file operations in the enriched environment.

Whether reads or writes dominate disk traffic varies significantly across workloads and environments. Based on these results, any general file system design must take into consideration the performance impact of both disk reads and disk writes.

#### 4.4.3 Effect of Memory Mapping

Another important factor in cache performance is the

TABLE 3. Process I/O

	INS	RES	WEB	NT
Processes that Read	209050 (10%)	103331 (12%)	8236 (9%)	1933 (36%)
Processes that Write	110008 (5%)	80426 (9%)	18505 (19%)	1182 (22%)
Processes that Memory Map	1525704 (72%)	584465 (68%)	37466 (39%)	4609 (85%)

Processes are tracked via `fork` and `exit` system calls. For all workloads, more processes use memory-mapped files than read or write. Because the NT traces do not continuously record all fork and exit information, the NT results are based on a subset of the traces.

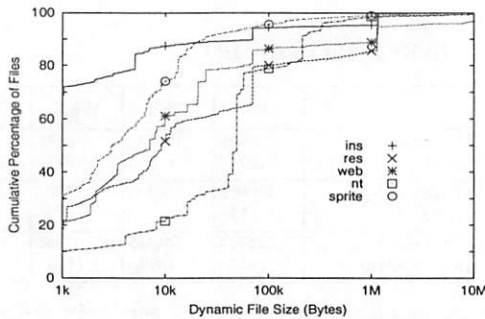
TABLE 4. Memory-mapped File Usage

	INS	RES	WEB
Avg. Mapped Files	43.4	17.6	7.4
Max. Mapped Files	91	47	10
Avg. Cache Space	23.2 MB	7.6 MB	2.4 MB
Max. Cache Space	41.2 MB	19.2 MB	3.0 MB
Cache Miss Rate	0.5%	1.5%	1.0%

For this data, each host maintains its own (unlimited size) cache of memory-mapped files, and only processes active on that host can affect the cache.

effect of memory-mapped files. Over the last few years, memory mapping has become a common method to access files, especially shared libraries. To see the impact of memory mapping on process I/O, we counted the number of processes that memory-map files and the number that perform reads and writes. Table 3 summarizes these results. For all workloads, a greater number of processes memory-map files than perform reads or writes. With such a high number of processes accessing memory-mapped files, people designing or evaluating file systems should not ignore the effect of these files on the I/O system.

Because our traces only monitor calls to map and unmap files, we do not have information on how programs access these files. For example, the traces do not indicate which parts of a mapped file the program accesses via memory loads. Although we do not have the precise access patterns, we estimate the effect of memory mapped files on the cache based on process calls to `mmap`, `munmap`, `fork`, and `exit`. Unfortunately, because our traces do not contain a complete record for forks and exits for the NT workload, we cannot perform an accurate estimate for the NT workload. For the UNIX workloads, we estimated the effect of memory-mapped files on the cache by keeping a list of all files that are



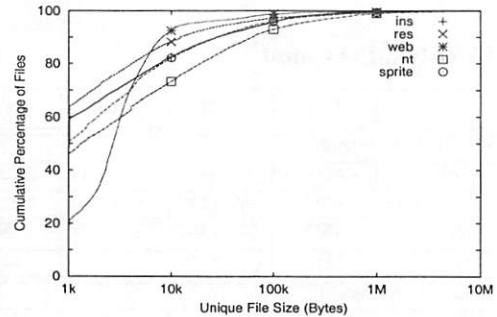
**FIGURE 6. Dynamic File Size.** We record file size for each accessed file when it is closed. If a file is opened and closed multiple times, we include the file in the graph data multiple times. Points depict sizes of 10KB, 100KB, and 1MB.

mapped either explicitly through a call to `mmap` or implicitly when a forked process inherits a file descriptor to a mapped file. We remove files from the list when no processes have the file mapped. Considering the number of `mmap` system calls, the average number of mapped files is quite low. The average and maximum number of files is shown in Table 4, along with the average and maximum space that would be required to keep the entire files in memory. We found that the same files tend to be mapped by many processes simultaneously. In fact, if the system kept each file in memory as long as at least one process mapped it, then cache miss rates for requests to map a file would only be about 1%.

## 4.5 File Size

Knowing the distribution of file sizes is important for designing metadata structures that efficiently support the range of file sizes commonly in use. The Sprite study found that most accessed files were small, but that the size of the largest files had increased since the BSD study. Our results show that this trend has continued.

In Figure 6, we show the file sizes across our workloads. In this graph, file size is determined *dynamically*—that is, file size is recorded for files as they are closed. With this methodology (also used in the Sprite study), files opened and closed multiple times are counted multiply. Like the Sprite study, we find that small files still comprise a large number of file accesses. The percentage of dynamically accessed files that are under 16KB is 88% for INS, 60% for RES, 63% for WEB, 24% for NT, and 86% for Sprite. At the other end of the spectrum, the number of accesses to large files has increased since the Sprite study. The number of files over 100KB accessed in Sprite is 4%, for INS it is 6%, for RES it is 20%, for WEB it is 14%, and for NT it is 21%. The largest file accessed in the Sprite traces is 38MB; the largest files in



**FIGURE 7. Unique File Size.** We record file size at the time of file close. If a file is opened and closed multiple times, we only use the last such event in the graph data. Points depict sizes of 10KB, 100KB, and 1MB.

the other traces are an order of magnitude larger: from 244MB (WEB) to 419MB (INS and NT).

In addition to dynamic file size distribution, we examined unique file size distribution. By this we mean a distribution computed by counting each file that occurs in the trace only once. Of course, this does not include any files that are never accessed, since they are not recorded in the traces. This distribution reflects the range of file sizes stored on disk that are actively accessed. Figure 7 shows the results. Assuming a disk block size of 8KB and an inode structure with twelve direct data pointers, files over 96KB must use indirect pointers. The percentage of files over 96KB is 4% for INS, 3% for RES, 1% for WEB, 7% for NT, and 4% for Sprite.

The WEB workload has many files in the 1–10KB range. A large number of these are image files. Because these images are exported over the Internet, the WEB administrators limit the size of these files to keep access latency small. Except for the NT workload, the unique file size distribution has not become more skewed towards larger files since the time of Sprite. Although the NT traces are two years younger than the UNIX traces, we believe its larger files are due to differences in the operating system and applications rather than the time difference since the six years between the UNIX and Sprite traces show no appreciable effect.

Although the size of the largest files has increased tenfold since the Sprite study, the unique file distribution indicates that, except for the NT workload, the percentage of large files has not increased since the Sprite study. However, the dynamic distribution indicates that large files are accessed a greater percentage of the time. As a result, the number of file accesses that require indirect pointers has increased. Since this trend is likely to continue, it may be worthwhile to redesign the inode struc-

ture to more efficiently support access to large files. However, since most files are still small, their data structures must still efficiently handle file sizes for a broad spectrum of sizes. File systems that use extent-based or multiple block sizes [Powe77] [Hitz94] may be more efficient at handling the range of file sizes in use today.

## 4.6 File Access Patterns

In this section, we examine file access patterns—that is, whether a file is read or written and the order in which its bytes are accessed. Knowing common access patterns is crucial to optimizing file system performance. For example, knowing that most files are read in their entirety, many file systems implement a simple prefetching strategy that prefetches blocks in sequential order.

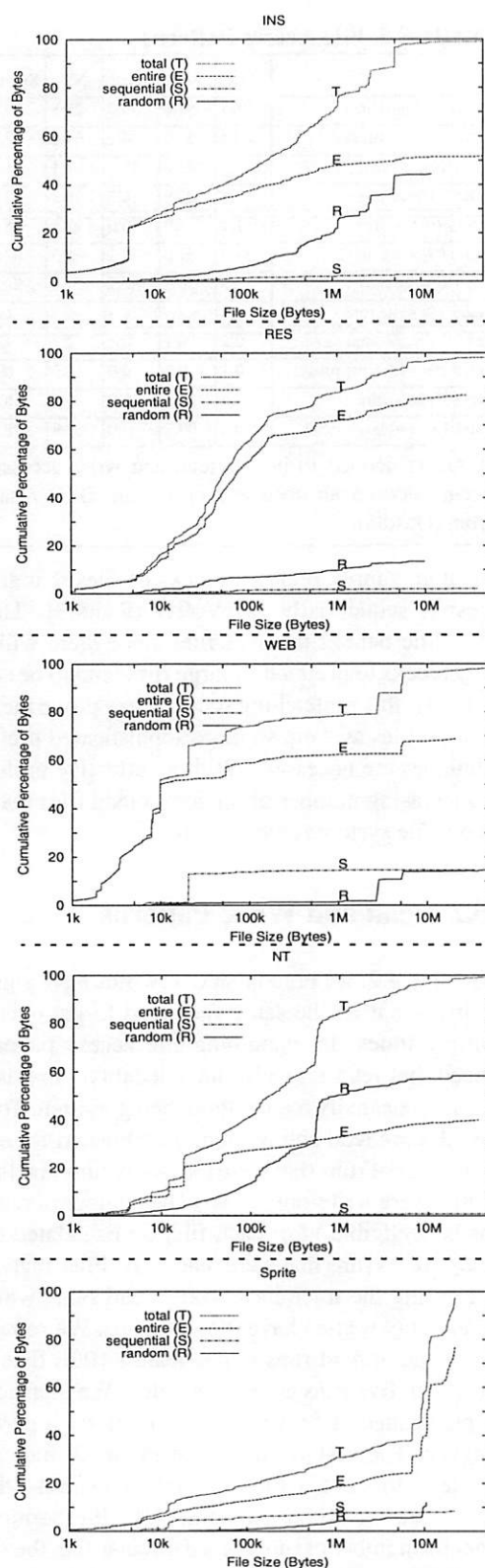
### 4.6.1 Run Patterns

We define a *run* as the accesses to a file that occur between its open and close. We classify runs into three categories. We classify a run as *entire* if it reads or writes a file once in order from beginning to end, *sequential* if it accesses the file sequentially but not from beginning to end, and *random* otherwise.

Table 5 compares file access patterns across workloads. Like Sprite and BSD, the majority of runs are reads and only a small percentage of runs contain both reads and writes. Also like the previous studies, most files are read in their entirety and most write runs are either entire or sequential. However, a higher percentage of runs are read-only in the HP-UX workloads than in NT, Sprite, or BSD. Also, our workloads tend to have a larger percentage of random reads than Sprite or BSD (the only exception being that BSD has a higher percentage of random runs than INS).

We examined random read patterns more closely and discovered a correlation between read pattern and file size. In Figure 8, we show the number of bytes transferred in entire, sequential, and random runs versus the size of the file being accessed. The graphs show that files that are less than 20KB are typically read in their entirety. For the Sprite workload, nearly all bytes are transferred in entire runs—even from very large files. However, for our workloads, large files tend to be read randomly. For INS, WEB, and NT, the majority of bytes from files over 100KB are accessed randomly. For RES, both entire runs and random runs are well-represented in bytes read from large files.

Most file systems are designed to provide good performance for sequential access to files. Prefetching strate-



**FIGURE 8. File Read Pattern versus File Size.** In these graphs, we plot the cumulative percentage of all bytes transferred versus file size for all transferred bytes, those transferred in entire runs, those transferred in sequential runs, and those transferred in random runs.



**TABLE 5. File Access Patterns**

	INS	RES	WEB	NT	Sprite	BSD
Reads (% total runs)	98.7	91.0	99.7	73.8	83.5	64.5
Entire (% read runs)	86.3	53.0	68.2	64.6	72.5	67.1
Seq. (% read runs)	5.9	23.2	17.5	7.1	25.4	24.0
Rand. (% read runs)	7.8	23.8	14.3	28.3	2.1	8.9
Writes (% total runs)	1.1	2.9	0.0	23.5	15.4	27.5
Entire (% write runs)	84.7	81.0	32.1	41.6	67.0	82.5
Seq. (% write runs)	9.3	16.5	66.1	57.1	28.9	17.2
Rand. (% write runs)	6.0	2.5	1.8	1.3	4.0	0.3
Read-Write (% total runs)	0.2	6.1	0.3	2.7	1.1	7.9
Entire (% read-write runs)	0.1	0.0	0.0	15.9	0.1	NA
Seq. (% read-write runs)	0.2	0.3	0.0	0.3	0.0	NA
Rand. (% read-write runs)	99.6	99.7	100	83.8	99.9	75.1

A run is defined to be the read and write accesses that occur between an open and close pair. BSD results are from [Oust85].

gies often simply prefetch blocks of files that are being accessed sequentially [McVo91] [Sand85]. This provides little benefit to small files since there will not be many blocks to prefetch. If large files tend to be accessed randomly, this prefetching scheme may prove ineffective for large files as well, so more sophisticated prefetching techniques are necessary. Without effective prefetching, the increasing number of randomly read files may result in poor file system response time.

#### 4.6.2 Read and Write Patterns

In Section 4.2, we noted that overwrites have significant locality—that is, the same files tend to get overwritten multiple times. In examining file access patterns, we noticed that read runs also have locality—that is, many files are repeatedly read without being written. To clarify how files are read and written, we tabulated for each file the number of runs that were read-only runs and the number that were write-only runs. (The number of read-write runs is negligible.) For each file, we calculated the percentage of its runs that were read-only. Files that are only read during the trace have 100% read runs, while files that are only written have 0% read runs. We rounded the percentage of read-runs to the nearest 10%; files having fewer than five runs are not included. We then added up the percentage of files that occurred in each percentage category. The results, shown in Figure 9, indicate that files tend to have a bimodal access pattern—they are either read-mostly or write-mostly. Furthermore, the larger the number of runs for a particular file, the stronger the affiliation. Many files tend to be read-mostly. This is evidenced by the large percentage of files that have 100% read runs. A small number of files are write-mostly. This is shown by the slight rise in the graphs at the 0% read-

only point. Note that while the percentage of files in this category is small, these files have many runs each. Files that are both read and written have a read-run percentage between 0% and 100%; however, as the number of runs increases, fewer files fall into these middle categories.

## 5 Conclusions

We collected file system traces from several different environments, consisting of an instructional workload, a research workload, a web workload, and a Windows NT personal computer workload. We used these traces to compare the file system behavior of these systems to each other and to systems studied in past research. Based on this analysis, we draw the following conclusions.

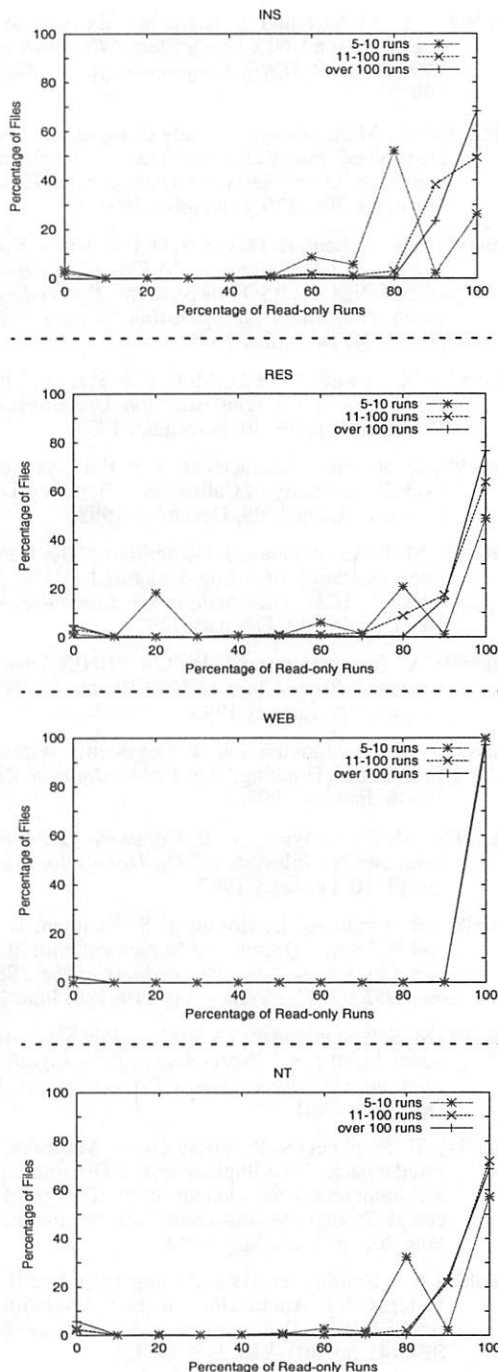
First, different systems show different I/O load. The WEB workload has far more read bandwidth than any other workload but has relatively little write bandwidth. The NT workload has more than twice the read and write bandwidth as the other workloads.

Second, we found that average block lifetime, and even the distribution of block lifetimes, varies significantly across workloads. In the UNIX workloads, most newly created blocks die within an hour. In contrast, in the NT workload, newly created blocks that survive one second are likely to remain alive over a day. However, common to all workloads are that 1) overwrites cause the most significant fraction of deleted blocks, and 2) overwrites show substantial locality. Due to this locality, a small write buffer is sufficient to absorb write traffic for nearly all workloads. What differs from one workload to another is the ideal write delay: some workloads perform well with the standard 30-second write delay while others benefit from a slightly longer delay.

Third, we examined the effect of caching on read traffic. We found that even small caches can sharply decrease disk read traffic. However, our results do not support the claim that disk traffic is dominated by writes when large caches are employed. Whether this claim holds depends not only on the cache size, but also on the workload and write delay.

Fourth, we determined that all modern workloads use memory-mapping to a large extent. We examined how memory-mapping is used in the UNIX workloads and found that a small number of memory-mapped files are shared among many active processes. From this we conclude that if each file were kept in memory as long as it is memory-mapped by any process, the miss rate for file map requests would be extremely low.





**FIGURE 9. Percentage of Runs that are Read-only.** Each line represents files categorized by the number of runs seen in the traces, where a run is defined to be all bytes transferred between the file's open and its close. The x-axis shows the percentage of runs that are read-only rounded to the nearest 10 percent. For each line, the percentages across the x-axis add to 100. Because most runs are read-mostly, the percentages are highest at the 100 percent read point, especially for files with many runs. A smaller number of files are write-mostly. These files appear at the 0 percent read runs point on the x-axis.

Fifth, we found that applications are accessing larger files than previously, and the maximum file size has increased in recent years. This is not surprising, as past studies have seen increases in file sizes as years passed. It might seem that increased accesses to large file sizes would lead to greater efficacy for simple readahead prefetching; however, we found that larger files are more likely to be accessed randomly than they used to be, rendering such straightforward prefetching less useful.

Finally, we found that for all workloads, file access patterns are bimodal in that most files tend to be mostly-read or mostly-written. We found this tendency to be especially strong for the files that are accessed most frequently. We expect file systems can make use of this knowledge to predict future file access patterns and optimize layout and access strategies accordingly.

## Acknowledgments

Trace collection is a major undertaking accomplished with the aid of many of people. For the UNIX traces, Francisco Lopez provided invaluable assistance as the local HP-UX guru and as the main technical support contact for the instructional cluster; Eric Fraser created the infrastructure for the trace collection host; Kevin Mullally, Jeff Anderson-Lee, Ginger Ogle, Monesh Sharma, and Michael Short contributed to the installation and testing of the trace collection process. For the NT traces, we would like to thank Alan Jay Smith, Jee Pang, John Vert, Felipe Cabrera, Patrick Franklin, Gordon Chaffee, and Windsor Hsu for help in designing and testing the tracer.

We would especially like to thank the anonymous users from the University of California at Berkeley, FP Enterprises, Toshiba Corporation, and the Michigan State Police, who let us run our tracer on their work machines. Without their participation, this work would have been impossible.

For providing valuable feedback on this work, we would like to gratefully acknowledge Jeanna Matthews, Adam Costello, Eric Anderson, Keith Smith, Thomas Kroeger, and Steve Lumetta.

## Trace Availability

The UNIX traces used for this paper are publicly available at <http://tracehost.cs.berkeley.edu/traces.html>.

## References

- [Bake91] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout, "Measurements of a Distributed File System," *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pp. 198–212, December 1991.
- [Bake92] M. Baker and M. Sullivan, "The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment," *Proceedings of the 1992 Summer USENIX Conference*, pp. 31–41, June 1992.
- [Benn91] J. M. Bennett, M. Bauer, and D. Kinchlea, "Characteristics of Files in NFS Environments," *Proceedings of the 1991 Symposium on Small Systems*, pp. 33–40, June 1991.
- [Blaz92] M. Blaze, "NFS Tracing by Passive Network Monitoring," *Proceedings of the 1992 Winter USENIX Conference*, pp. 333–343, January 1992.
- [Bozm91] G. Bozman, H. Ghannad, and E. Weinberger, "A Trace-Driven Study of CMS File References," *IBM Journal of Research and Development*, 35(5–6), pp. 815–828, September–November 1991.
- [Chen96] P. Chen, W. Ng, S. Chandra, C. Aycok, G. Rajamani, and D. Lowell, "The Rio File Cache: Surviving Operating System Crashes," *Proceedings of the Seventh ASPLOS Conference*, pp. 74–83, October 1996.
- [Chia93] C. Chiang and M. Mutka, "Characteristics of User File-Usage Patterns," *Systems and Software*, 23(3), pp. 257–268, December 1993.
- [Dahl94] M. Dahlin, C. Mather, R. Wang, T. Anderson, and D. Patterson, "A Quantitative Analysis of Cache Policies for Scalable Network File Systems," *Proceedings of the 1994 Sigmetrics Conference*, pp. 150–160, May 1994.
- [Douc99] J. Douceur and W. Bolosky, "A Large-Scale Study of File-System Contents," *Proceedings of the 1999 Sigmetrics Conference*, pp. 59–70, June 1999.
- [Floy89] R. Floyd and C. Schlatter Ellis, "Directory Reference Patterns in Hierarchical File Systems," *IEEE Transactions on Knowledge and Data Engineering*, 1(2), pp. 238–247, June 1989.
- [Gang97] G. Ganger and M. F. Kaashoek, "Embedded Inodes and Explicit Groupings: Exploiting Disk Bandwidth for Small Files," *Proceedings of the USENIX Annual Technical Conference*, pp. 1–17, January 1997.
- [Grib98] S. Gribble, G. Manku, D. Roselli, E. Brewer, T. Gibson, and E. Miller, "Self-Similarity in File Systems," *Proceedings of the 1998 Sigmetrics Conference*, pp. 141–150, June 1998.
- [Hart93] J. Hartman and J. Ousterhout, "Corrections to Measurements of a Distributed File System," *Operating Systems Review*, 27(1), pp. 7–10, January 1993.
- [Hitz94] D. Hitz, J. Lau, M. Malcolm, "File System Design for an NFS File Server Appliance," *Proceedings of the 1994 Winter USENIX Conference*, pp. 235–246, January 1994.
- [Lorc00] J. Lorch and A. J. Smith, "Building VTrace, a Tracer for Windows NT," Accepted for publication in *MSDN Magazine*, September–October 2000.
- [McKu84] M. McKusick, W. Joy, S. Leffler, and R. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, 2(3), pp. 181–197, August 1984.
- [McVo91] L. McVoy and S. Kleiman, "Extent-like Performance from a UNIX File System," *Proceedings of the 1991 Winter USENIX Conference*, pp. 33–44, January 1991.
- [Mumm94] L. Mummert and M. Satyanarayanan, "Long-term Distributed File Reference Tracing: Implementation and Experience," *Software—Practice and Experience*, 26(6), pp. 705–736, November 1994.
- [Oust85] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," *Proceedings of the Tenth Symposium on Operating Systems Principles*, pp. 15–24, December 1985.
- [Powe77] M. Powell, "The DEMOS File System," *Proceedings of the Sixth Symposium on Operating Systems Principles*, pp. 39–40, November 1977.
- [Rose98] D. Roselli, "Characteristics of File System Workloads," University of California at Berkeley Technical Report CSD-98-1029, December 1998.
- [Rose92] M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-Structured File System for UNIX," *ACM Transactions on Computer Systems*, 10(1), pp. 26–52, February 1992.
- [Ruem93] C. Ruemmler and J. Wilkes, "UNIX Disk Access Patterns," *Proceedings of 1993 Winter USENIX Conference*, CA, January 1993.
- [Russ97a] M. Russinovich and B. Cogswell, "Windows NT System-Call Hooking," *Dr. Dobbs's Journal*, 22(1), pp. 42–46, January 1997.
- [Russ97b] M. Russinovich and B. Cogswell, "Examining the Windows NT Filesystem," *Dr. Dobbs's Journal*, 22(2), pp. 42–50, February 1997.
- [Sand85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network Filesystem," *Proceedings of the 1985 Summer USENIX Conference*, pp. 119–130, June 1985.
- [Saty81] M. Satyanarayanan, "A Study of File Sizes and Functional Lifetimes," *Proceedings of the Eighth Symposium on Operating System Principles*, pp. 96–108, December 1981.
- [Sien94] T. Sienknecht, R. Friedrich, J. Martinka, and P. Friedenbach, "The Implications of Distributed Data in a Commercial Environment on the Design of Hierarchical Storage Management," *Performance Evaluation*, 20, pp. 3–25, May 1994.
- [Smit81] A. J. Smith, "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms," *IEEE Transactions on Software Engineering*, SE-7(4), pp. 403–416, July 1981.
- [Voge99] W. Vogels, "File System Usage in Windows NT 4.0," *Proceedings of the Seventeenth Symposium on Operating Systems Principles*, pp. 93–109, December 1999.
- [Zhou99] M. Zhou and A. J. Smith, "Analysis of Personal Computer Workloads," *Proceedings of the Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 208–217, October 1999.

# FiST: A Language for Stackable File Systems

Erez Zadok and Jason Nieh

Computer Science Department, Columbia University

{ezk,nieh}@cs.columbia.edu

## Abstract

Traditional file system development is difficult. Stackable file systems promise to ease the development of file systems by offering a mechanism for incremental development. Unfortunately, existing methods often require writing complex low-level kernel code that is specific to a single operating system platform and also difficult to port.

We propose a new language, *FiST*, to describe stackable file systems. *FiST* uses operations common to file system interfaces. From a single description, *FiST*'s compiler produces file system modules for multiple platforms. The generated code handles many kernel details, freeing developers to concentrate on the main issues of their file systems.

This paper describes the design, implementation, and evaluation of *FiST*. We extended file system functionality in a portable way without changing existing kernels. We built several file systems using *FiST* on Solaris, FreeBSD, and Linux. Our experiences with these examples shows the following benefits of *FiST*: average code size over other stackable file systems is reduced ten times; average development time is reduced seven times; performance overhead of stacking is 1–2%.

## 1 Introduction

File systems have proven to be useful in enriching system functionality. The abstraction of folders with files containing data is natural for use with existing file browsers, text editors, and other tools. Modifying file systems became a popular method of extending new functionality to users. However, developing file systems is difficult and involved. Developers often use existing code for native in-kernel file systems as a starting point[15, 23]. Such file systems are difficult to write and port because they depend on many operating system specifics, and they often contain many lines of complex operating systems code, as seen in Table 1.

User-level file systems are easier to develop and port because they reside outside the kernel[16]. However, their

Media Type	Common File System	Avg. Code Size (C lines)
Hard Disks	UFS, FFS, EXT2FS	5,000–20,000
Network	NFS	6,000–30,000
CD-ROM	HSFS, ISO-9660	3,000–6,000
Floppy	PCFS, MS-DOS	5,000–6,000

Table 1: Common Native Unix File Systems and Code Sizes for Each Medium

performance is poor due to the extra context switches these file systems must incur. These context switches can affect performance by as much as an order of magnitude[26, 27].

*Stackable file systems*[19] promise to speed file system development by providing an extensible file system interface. This extensibility allows new features to be added incrementally. Several new extensible interfaces have been proposed and a few have been implemented[8, 15, 18, 22]. To improve performance, these stackable file systems were designed to run in the kernel. Unfortunately, using these stackable interfaces often requires writing lots of complex C kernel code that is specific to a single operating system platform and also difficult to port.

More recently, we introduced a stackable template system called *Wrapfs*[27]. It eases up file system development by providing some built-in support for common file system activities. It also improves portability by providing kernel templates for several operating systems. While working with *Wrapfs* is easier than with other stackable file systems, developers still have to write kernel C code and port it using the platform-specific templates.

In previous approaches, performance and portability could not be achieved together. To perform well, a file system should run in the kernel, not at user level. Kernel code, however, is much more difficult to write and port than user-level code. To ease the problems of developing and porting stackable file systems that perform well, we propose a high-level language to describe such file systems. There are three benefits to using a language:



1. **Simplicity:** A file system language can provide familiar higher-level primitives that simplify file system development. The language can also define suitable defaults automatically. These reduce the amount of code that developers need to write, and lessen their need for extensive knowledge of kernel internals, allowing even non-experts to develop file systems.
2. **Portability:** A language can describe file systems using an interface abstraction that is common to operating systems. The language compiler can bridge the gaps among different systems' interfaces. From a single description of a file system, we could generate file system code for different platforms. This improves portability considerably. At the same time, however, the language should allow developers to take advantage of system-specific features.
3. **Specialization:** A language allows developers to customize the file system to their needs. Instead of having one large and complex file system with many features that may be configured and turned on or off, the compiler can produce special-purpose file systems. This improves performance and memory footprint because specialized file systems include only necessary code.

This paper describes the design and implementation of *FiST*, a *File System Translator* language for stackable file systems. *FiST* lets developers describe stackable file systems at a high level, using operations common to file system interfaces. With *FiST*, developers need only describe the core functionality of their file systems. The *FiST* language code generator, *fistgen*, generates kernel file system modules for several platforms using a single description. We currently support Solaris, FreeBSD, and Linux.

To assist *fistgen* with generating stackable file systems, we created a minimal stackable file system template called *Basefs*. *Basefs* adds stacking functionality missing from systems and relieves *fistgen* from dealing with many platform-dependent aspects of file systems. *Basefs* does not require changes to the kernel or existing file systems. Its main function is to handle many kernel details relating to stacking. *Basefs* provides simple hooks for *fistgen* to insert code that performs common tasks desired by file system developers, such as modifying file data or inspecting file names. That way, *fistgen* can produce file system code for any platform we port *Basefs* to. The hooks also allow *fistgen* to include only necessary code, improving performance and reducing kernel memory usage.

We built several example file systems using *FiST*. Our experiences with these examples shows the following benefits of *FiST* compared with other stackable file systems: average code size is reduced ten times; development time is reduced seven times; performance overhead of stacking is less than 2%, and unlike other stacking systems, there is no performance overhead for native file systems.

Our focus in this paper is to demonstrate how *FiST* simplifies the development of file systems, provides write-once run-anywhere portability across UNIX systems, and reduces stacking overhead through file system specialization. The rest of this paper is organized as follows. Section 2 details the design of *FiST*, and describes the *FiST* language, *fistgen*, and *Basefs*. Section 3 discusses key implementation and portability details. Section 4 describes several example file systems written using *FiST*. Section 5 evaluates the ease of development, the portability, and the performance of our file systems. Section 6 surveys related work. Finally, Section 7 concludes and explores future directions.

## 2 Design

*FiST* is a high level language providing a file system abstraction. Figure 1 shows the hierarchy for different file system abstractions. At the lowest level reside file systems native to the operating system, such as disk based and network based file systems. They are at the lowest level because they interact directly with device drivers. Above native file systems are stackable file systems such as the examples in Section 4, as well as *Basefs*. These file systems provide a higher abstraction than native file systems because stackable file systems interact only with other file systems through a well defined *virtual file system interface* (VFS)[11]. The VFS provides *virtual nodes* (vnodes), an abstraction of files across different file systems. However, both these levels are system specific.

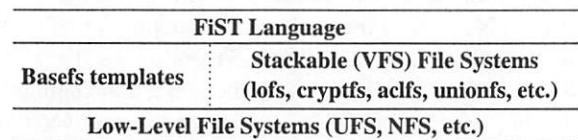


Figure 1: *FiST* Structural Diagram. Stackable file systems, including *Basefs*, are at the VFS level, and are above low-level file systems. *FiST* descriptions provide a higher abstraction than that provided by the VFS.

At the highest level, we define the *FiST* language. *FiST* abstracts the different vnode interfaces across *different* operating systems into a single common description language, because it is easier to write file systems this way. We found that while vnode interfaces differ from system to system, they share many similar features. Our experience shows that similar file system concepts exist in other non-Unix systems, and our stacking work can be generalized to include them. Therefore, we designed the *FiST* language to be as general as possible: we mirror existing platform-specific vnode interfaces, and extend them through the *FiST* language in a platform independent way. This allows us to modify vnode operations and the arguments they pass in an arbitrary way, providing great design flexibility. At the same time, this abstraction means that stackable



file systems cannot easily access device drivers and control, for example, block layout of files on disks and the existing structure of meta-data (inodes).

FiST does not require that applications be changed. The default behavior of produced code maintains compatibility with existing file system APIs. FiST does allow, however, the creation of special-purpose file systems that can extend new functionality to applications.

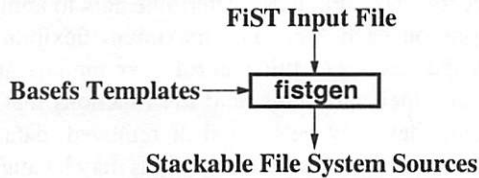


Figure 2: FiST Operational Diagram. *Fistgen* reads a FiST input file, and with the Basefs templates, produces sources for a new file system.

The overall operation of the FiST system is shown in Figure 2. The figure illustrates how the three parts of FiST work together: the FiST language, *fistgen*, and Basefs. File system developers write FiST input files to implement file systems using the FiST language. *Fistgen*, the FiST language code parser and file system code generator, reads FiST input files that describe the new file system's functionality. *Fistgen* then uses additional input files, the Basefs templates. These templates contain the stacking support code for each operating system and hooks to insert developer code. *Fistgen* combines the functionality described in the FiST input file with the Basefs templates, and produces new kernel C sources as output. The latter implement the functionality of the new file system. Developers can, for example, write simple FiST code to manipulate file data and file names. *Fistgen*, in turn, translates that FiST code into C code and inserts it at the right place in the templates, along with any additional support code that may be required. Developers can also turn on or off certain file system features, and *fistgen* will conditionally include code that implements those features.

## 2.1 A Quick Example: Snoopfs

To illustrate the FiST development process, we contrast it with traditional file system development methods using a simple example similar to Watchdogs[2]. Suppose a file system developer wants to write a file system that will warn of any possible unauthorized access to users' files. The main idea is that only the files' owner or the root user are allowed access. Any other user who might be attempting to find files that belong to another user, would normally get a "permission denied" error code. However, the system does not produce an alert when such an attempt is made. This new *snooping file system* (Snoopfs) will log these failed attempts.

The one place where such a check should be made is in the lookup routine that is used to find a file in a directory. To do so without FiST, the developer has to do the following:

1. locate an operating system with available sources for one file system
2. read and understand the code for that file system and any associated kernel code
3. make a copy of the sources, and carefully modify them to include the new functionality
4. compile the sources into a new file system, possibly rebuilding a new kernel and rebooting the system
5. mount the new file system, test, and debug as needed

After completing this, the developer is left with one modified file system for one operating system. The amount of code that has to be read and understood ranges in the thousands of lines (Table 1). The process has to be repeated for each new port to a new platform. In addition, changes to native file system are unlikely to be accepted by operating system maintainers, and have to be maintained independently.

In contrast, the normal procedure for developing code with FiST is:

1. write the code in FiST once
2. run *fistgen* on the input file
3. compile the produced sources into a loadable kernel module, and load it into a running system
4. mount the new file system, test, and debug as needed

Debugging code can be turned on in FiST to assist in the development of the new file system. There is no need to have kernel sources or be familiar with them; there is no need to write or port code for each platform; and there is no need to rebuild or reboot the kernel. Furthermore, the same developer can write Snoopfs using a small number of lines of FiST code:

```

%op:lookup:postcall {
if ((fistLastErr() == EPERM ||
    fistLastErr() == ENOENT) &&
    $0.owner != %uid && %uid != 0)
    fistPrintf("snoopfs detected access by uid %d,\n
pid %d, to file %s\n", %uid, %pid, $name);
}
  
```

This short FiST code inserts an "if" statement after the normal call to the lookup routine. The code checks if the previous lookup call failed with one of two particular errors, who the owner of the directory is, who the effective running user is, and then decides whether to print the warning message.

This single FiST description is portable, and can be compiled on each platform that we have ported our templates to (currently three).

## 2.2 The File System Model

A FiST-produced file system runs in the kernel, as seen in Figure 3. FiST file systems mirror the vnode interface both above and below. The interface to user processes is the system call interface. FiST does not change either the system call interface or the vnode interface. Instead, FiST can change information passed and returned through the interfaces.

A user process generally accesses a file system by executing a system call, which traps into the kernel. The kernel VFS then translates the system call to a vnode operation, and calls the corresponding file system. If the latter is a FiST-produced file system, it may call another stacked file system below. Once the execution flow has reached the lowest file system, error codes and return values begin flowing upwards, all the way to the user process.

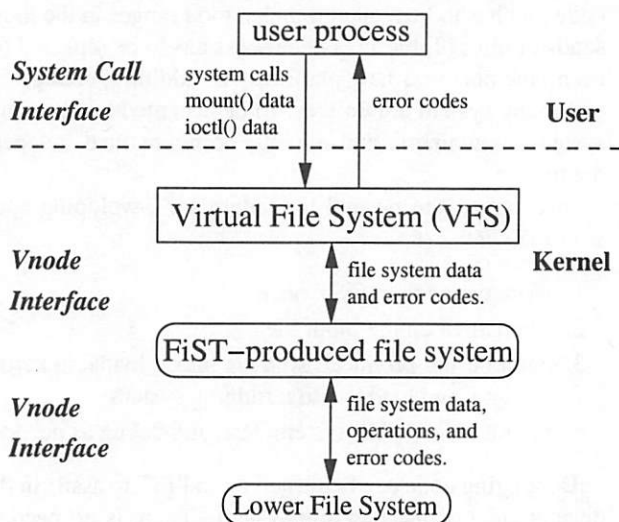


Figure 3: Information and execution flow in a stackable system. FiST does not change the system call or vnode interfaces, but allows for arbitrary data and control operations to flow in both directions.

In FiST, we model a file system as a collection of mounts, files, and user processes, all running under one system. Several *mounts*, mounted instances of file systems, can exist at any time. A FiST-produced file system can access and manipulate various mounts and files, data associated with them, their attributes, and the functions that operate on them. Furthermore, the file system can access attributes that correspond to the run-time execution environment: the operating system and the user process currently executing.

Information (both data and control) generally flows between user processes and the mounted file system through the system call interface. For example, file data flows between user processes and the kernel via the `read` and `write` system calls. Processes can pass specific file

system data using the `mount` system call. In addition, mounted file systems may return arbitrary (even new) error codes back to user processes.

Since a FiST-produced stackable file system is the caller of other file systems, it has a lot of control over what transpires between it and the ones below through the vnode interface. FiST allows access to multiple mounts and files. Each mount or file may have multiple attributes that FiST can access. Also, FiST can determine how to apply vnode functions on each file. For maximum flexibility, FiST allows the developer full control over mounts and files, their data, their attributes, and the functions that operate on them; they may be created or removed, data and attributes can be changed, and functions may be augmented, replaced, reordered, or even ignored.

Ioctl (I/O Controls) have been used as an operating system extension mechanism as they can exchange arbitrary information between user processes and the kernel, as well as in between file system layers, without changing interfaces. FiST allows developers to define new ioctls and define the actions to take when they are used; this can be used to create application-specific file systems. FiST also provides functions for portable copying of ioctl data between user and kernel spaces. For example, our encryption file system (Section 4.1) uses an ioctl to set cipher keys.

Traditional stackable file systems create a single linear stack of mounts, each one hiding the one file system below it. More general stacking allows for a tree-like mount structure, as well as for direct access to any layer [8, 18]. This interesting aspect of stackable file systems is called *fanning*, as shown in Figure 4. A *fan-out* allows the mounted file system to access two or more mounts below. A fan-out is useful for example in replicated, load-balancing, unifying [15], or caching file systems [22].

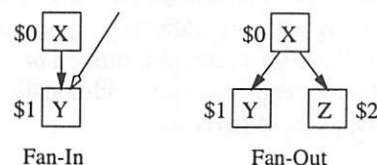


Figure 4: Fanning in stackable file systems

A *fan-in* allows a process to access lower level mounts directly. This can be useful when fast access to the lower level data is needed. For example, in an encryption file system, a backup utility can backup the data faster (and more securely) by accessing the ciphertext files in the lower level file system. If fan-in is not used, the mounted file system will overlay the mounted directory with the mount point. An overlay mount hides the lower level file system. This can be useful for some security applications. For example, our ACL file system (Section 4.2) hides certain important files from normal view and is able to control who can manipulate those files and how.

## 2.3 The FiST Language

The FiST language is a high-level language that uses file system features common to several operating systems. It provides file system specific language constructs for simplifying file system development. In addition, FiST language constructs can be used in conjunction with additional C code to offer the full flexibility of a system programming language familiar to file system developers. The ability to integrate C and FiST code is reflected in the general structure of FiST input files. Figure 5 shows the four main sections of a FiST input file.

	%{
1	<i>C Declarations</i>
	%}
2	<i>FiST Declarations</i>
	%%
3	<i>FiST Rules</i>
	%%
4	<i>Additional C Code</i>

Figure 5: FiST Grammar Outline

The FiST grammar was modeled after yacc[9] input files, because yacc is familiar to programmers and the purpose for each of its four sections (delimited by “%%”) matches with four different subdivisions of desired file system code: raw included header declarations, declarations that affect the produced code globally, actions to perform when matching vnode operations, and additional code.

**C Declarations** (enclosed in “{% %}”) are used to include additional C headers, define macros or typedefs, list forward function prototypes, etc. These declarations are used throughout the rest of the code.

**FiST Declarations** define global file system properties that affect the overall semantics of the produced code and how a mounted file system will behave. These properties are useful because they allow developers to make common global changes in a simple manner. In this section we declare if the file system will be read-only or not, whether or not to include debugging code, if fan-in is allowed or not, and what level (if any) of fan-out is used.

FiST Declarations can also define special data structures used by the rest of the code for this file system. We can define mount-time data that can be passed with the mount(2) system call. A versioning file system, for example, can be passed a number indicating the maximum number of versions to allow per file. FiST can also define new error codes that can be returned to user processes, for the latter to understand additional modes of failure. For example, an encryption file system can return a new error code indicating that the cipher key in use has expired.

**FiST Rules** define actions that generally determine the behavior for individual files. A FiST rule is a piece of code that executes for a selected set of vnode operations, for one

operation, or even a portion of a vnode operation. Rules allow developers to control the behavior of one or more file system functions in a portable manner. The FiST rules section is the primary section, where most of the actions for the produced code are written. In this section, for example, we can choose to change the behavior of `unlink` to rename the target file, so it might be restored later. We separated the declarations and rules sections for programming ease: developers know that global declarations go in the former, and actions that affect vnode operations go in the latter.

**Additional C Code** includes additional C functions that might be referenced by code in the rest of the file system. We separated this section from the rules section for code modularity: FiST rules are actions to take for a given vnode function, while the additional C code may contain arbitrary code that could be called from anywhere. This section provides a flexible extension mechanism for FiST-based file systems. Code in this section may use any basic FiST primitives (discussed in Section 2.3.1) which are helpful in writing portable code. We also allow developers to write code that takes advantage of system-specific features; this flexibility, however, may result in non-portable code.

The remainder of this section introduces the FiST language primitives, the various participants in a file system (such as files, mounts, and processes), their attributes and how to extend them and store them persistently, and how to control the execution flow in a file system. The examples in Section 4 are also helpful because they further illustrate the FiST language.

### 2.3.1 FiST Syntax

FiST syntax allows referencing mounted file systems and files, accessing attributes, and calling FiST functions. Mount references begin with `$vfs`, while file references use a shorter “\$” syntax because we expect them to appear more often in FiST code. References may be followed by a name or number that distinguishes among multiple instances (e.g., `$1`, `$2`, etc.) especially useful when fan-out is used (Figure 4). Attributes of mounts and files are specified by appending a dot and the attribute name to the reference (e.g., `$vfs.blocksize`, `$1.name`, `$2.owner`, etc.) The scope of these references is the current vnode function in which they are executing.

There is only one instance of a running operating system. Similarly, there is only one process context executing that the file system has to be concerned with. Therefore FiST need only refer to their attributes. These read-only attributes are summarized in Table 2. The scope of all read-only “%” attributes is global.

FiST code can call FiST functions from anywhere in the file system, some of which are shown in Table 3. The scope of FiST functions is global in the mounted file system. These functions form a comprehensive library of portable



Global	Meaning
%blocksize	native disk block size
%gid	effective group ID
%pagesize	native page size
%pid	process ID
%time	current time (seconds since epoch)
%uid	effective user ID

Table 2: Global Read-Only FiST Variables

routines useful in writing file systems. The names of these functions begin with “fist.” FiST functions can take a variable number of arguments, omit some arguments where suitable defaults exist, and use different types for each argument. These are true functions that can be nested and may return any single value.

Function	Meaning
fistPrintf	print messages
fistStrEq	string comparison
fistMemCpy	buffer copying similar
fistLastErr	get the last error code
fistSetErr	set the return error code
fistReturnErr	return an error code immediately
fistSetIoctlData	set ioctl value to pass to a user process
fistGetIoctlData	get ioctl value from a user process
fistSetFileData	write arbitrary data to a file
fistGetFileData	read arbitrary data from a file
fistLookup	find a file in a directory
fistReaddir	read a directory
fistSkipName	hide a name of a file in a directory
fistOp	execute an arbitrary vnode operation

Table 3: A sample of FiST functions used in this paper

Each mount and file has attributes associated with it. FiST recognizes common attributes of mounted file systems and files that are defined by the system, such as the name, owner, last modification time, or protection modes. FiST also allows developers to define new attributes and optionally store them persistently. Attributes are accessed by appending the name of the attribute to the mount or file reference, with a single dot in between, much the same way that C dereferences structure field names. For example, the native block size of a mounted file system is accessed as `$vfs.blocksize` and the name of a file is `$0.name`.

FiST allows users to create new file attributes. For example, an ACL file system may wish to add timed access to certain files. The following FiST Declaration can define the new file attributes in such a file system:

```
per_vnode {
    int    user;      /* extra user */
    int    group;     /* extra group */
    time_t expire;    /* access expiration time */
};
```

With the above definition in place, a FiST file system may refer to the additional user and group who are allowed

to access the file as `$0.user` and `$0.group`, respectively. The expiration time is accessed as `$0.expire`.

The `per_vnode` declaration defines new attributes for files, but those attributes are only kept in memory. FiST also provides different methods to define, store, and access additional attributes persistently. This way, a file system developer has the flexibility of deciding if new attributes need only remain in memory or saved more permanently.

For example, an encrypting file system may want to store an encryption key, cipher ID, and Initialization Vector (IV) for each file. This can be declared in FiST using:

```
fileformat SECDAT {
    char    key[16];   /* cipher key */
    int     cipher;    /* cipher ID */
    char    iv[16];    /* initialization vector */
};
```

Two FiST functions exist for handling file formats: `fistSetFileData` and `fistGetFileData`. These two routines can store persistently and retrieve (respectively) additional file system and file attributes, as well as any other arbitrary data. For example, to save the cipher ID in a file called `.key`, use:

```
int cid;
/* set cipher ID */
fistSetFileData(".key", SECDAT, cipher, cid);
```

The above FiST function will produce kernel code to open the file named `.key` and write the value of the “cid” variable into the “cipher” field of the “SECDAT” file format, as if the latter had been a data structure stored in the `.key` file.

Finally, the mechanism for adding new attributes to mounts is similar. For files, the declaration is `per_vnode` while for mounts it is `per_vfs`. The routines `fistSetFileData` and `fistGetFileData` can be used to access any arbitrary persistent data, for both mounts and files.

### 2.3.2 Rules for Controlling Execution and Information Flow

In the previous sections we considered how FiST can control the flow of information between the various layers. In this section we describe how FiST can control the execution flow of various operations using FiST rules.

FiST does not change the interfaces that call it, because such changes will not be portable across operating systems and may require changing many user applications. FiST therefore only exchanges information with applications using existing APIs (e.g., ioctls) and those specific applications can then affect change.

The most control FiST file systems have is over the file system (vnode) operations that execute in a normal stackable setting. Figure 6 highlights what a typical stackable vnode operation does: (1) find the vnode of the lower level



mount, and (2) repeat the same operation on the lower vnode.

```
int f$fname_getattr(vnode_t *vp, args...)
{
    int error;
    vnode_t *lower_vp = get_lower(vp);

    /* (1) pre-call code goes here */
    /* (2) call same operation on lower file system */
    error = VOP_GETATTR(lower_vp, args...);
    /* (3) post-call code goes here */
    return error;
}
```

Figure 6: A skeleton of typical kernel C code for stackable vnode functions. FiST can control all three sections of every vnode function: pre-call, post-call, and the call itself.

The example vnode function receives a pointer to the vnode on which to apply the operation, and other arguments. First, the function finds the corresponding vnode at the lower level mount. Next, the function actually calls the lower level mounted file system through a standard VOP\_\* macro that applies the same operation, but on the file system corresponding to the type of the lower vnode. The macro uses the lower level vnode, and the rest of the arguments unchanged. Finally, the function returns to the caller the status code which the lower level mount passed to the function.

There are three key parts in any stackable function that FiST can control: the code that may run before calling the lower level mount (pre-call), the code that may run afterwards (post-call), and the actual call to the lower level mount. FiST can insert arbitrary code in the pre-call and post-call sections, as well as replace the call part itself with anything else.

By default, the pre-call and post-call sections are empty, and the call section contains code to pass the operation to the lower level file system. These defaults produce a file system that stacks on another but does not change behavior, and that was designed so developers do not have to worry about the basic stacking behavior—only about their changes.

For example, a useful pre-call code in an encryption file system would be to verify the validity of cipher keys. A replication file system may insert post-call code to repeat the same vnode operation on other replicas. A versioning file system could replace the actual call to remove a file with a call to rename it; an example FiST code for the latter might be:

```
%op:unlink:call {
    fistRename($name, fistStrAdd($name, ".unrm"));
}
```

The general form for a FiST rule is:

*%callset : optype : part {code}* (1)

Table 4 summarizes the possible values that a FiST rule can have. *Callset* defines a collection of operations to operate on. *Optype* further defines the call set to a subset of operations or a single operation. *Part* defines the part of the call that the following code refers to: pre-call, call, post-call, or the name of a newly defined ioctl. Finally, *code* contains any C code enclosed in braces.

Call Sets	
op	to refer to a single operation
ops	to refer to all operations
readops	to refer to non state changing operations
writeops	to refer to state changing operations
Operation Types	
all	all operations
data	operations that manipulate file data
name	operations that manipulate file names
The rest of the operation types specify one of the following vnode operations: create, getattr, l/stat, link, lookup, mkdir, read, readdir, readlink, rename, rmdir, setattr, statfs, symlink, unlink, and write.	
Call Part	
precall	part before calling the lower file system
call	the actual call to the lower file system
postcall	part after calling the lower file system
ioctl	name of a newly defined ioctl

Table 4: Possible Values in FiST Rules

### 2.3.3 Filter Declarations and Filter Functions

FiST file systems can perform arbitrary manipulations of the data they exchange between layers. The most useful and at the same time most complex data manipulations in a stackable file system involve file data and file names. To manipulate them consistently without FiST or Wrapsfs, developers must make careful changes in many places. For example, file data is manipulated in read, write, and all of the MMAP functions; file names also appear in many places: lookup, create, unlink, readdir, mkdir, etc.

FiST simplifies the task of manipulating file data or file names using two types of *filters*. A filter is a function like Unix shell filters such as sed or sort: they take some input, and produce possibly modified output.

If developers declare *filter:data* in their FiST file, fistgen looks for two data coding functions in the Additional C Code section of the FiST File: *encode\_data* and *decode\_data*. These functions take an input data page, and an allocated output page of the same size. Developers are expected to implement these coding functions in the Additional C Code section of the FiST file. The two functions must fill in the output page by encoding or decoding

it appropriately and return a success or failure status code. Our encryption file system uses a data filter to encrypt and decrypt data (Section 4.1).

With the FiST declaration `filter:name`, `fistgen` inserts code and calls to encode or decode strings representing file names. The file name coding functions (`encode_name` and `decode_name`) take an input file name string and its length. They must allocate a new string and encode or decode the file name appropriately. Finally, the coding functions return the number of bytes in the newly allocated string, or a negative error code. `Fistgen` inserts code at the caller's level to free the memory allocated by file name coding functions.

Using FiST filters, developers can easily produce file systems that perform complex manipulations of data or names exchanged between file system layers.

## 2.4 `Fistgen`

`Fistgen` is the FiST language code generator. `Fistgen` reads in an input FiST file, and using the right Basefs templates, produces all the files necessary to build a new file system described in the FiST input file. These output files include C file system source files, headers, sources for user level utilities, and a Makefile to compile them on the given platform.

`Fistgen` implements a subset of the C language parser and a subset of the C preprocessor. It handles conditional macros (such as `#ifdef` and `#endif`). It recognizes the beginning of functions after the first set of declarations and the ending of functions. It parses FiST tags inserted in Basefs (explained in the next section) used to mark special places in the templates. Finally, `fistgen` handles FiST variables (beginning with `$` or `%`) and FiST functions (such as `fist-Lookup`) and their arguments.

After parsing an input file, `fistgen` builds internal data structures and symbol tables for all the keywords it must handle. `Fistgen` then reads the templates, and generates output files for each file in the template directory. For each such file, `fistgen` inserts needed code, excludes unused code, or replaces existing code. In particular, `fistgen` conditionally includes large portions of code that support FiST filters: code to manipulate file data or file names. It also produces several new files (including comments) useful in the compilation for the new file system: a header file for common definitions, and two source files containing auxiliary code.

The code generated by `fistgen` may contain automatically generated functions that are necessary to support proper FiST function semantics. Each FiST function is replaced with one true C function—not a macro, inlined code, a block of code statements, or any feature that may not be portable across operating systems and compilers. While it might have been possible to use other mechanisms such as

C macros to handle some of the FiST language, it would have resulted in unmaintainable and unreadable code. One of the advantages of the FiST system is that it produces highly readable code. Developers can even edit that code and add more features by hand, if they so choose.

`Fistgen` also produces real C functions for specialized FiST syntax that cannot be trivially handled in C. For example, the `fistGetIoctlData` function takes arguments that represent names of data structures and names of fields within. A C function cannot pass such arguments; C++ templates would be needed, but we opted against C++ to avoid requiring developers to know another language, because modern Unix kernels are still written in C, and to avoid interoperability problems between C++ produced code and C produced code in a running kernel. Preprocessor macros can handle data structure names and names of fields, but they do not have exact or portable C function semantics. To solve this problem, `fistgen` replaces calls to functions such as `fistGetIoctlData` with automatically generated specially named C functions that hard-code the names of the data structures and fields to manipulate. `Fistgen` generates these functions only if needed and only once.

## 2.5 Basefs

Basefs is a template system which was derived from `Wraps`[27]. It provides basic stacking functionality without changing other file systems or the kernel. To achieve this functionality, the kernel must support three features. First, in each of the VFS data structures, Basefs requires a field to store pointers to data structures at the layer below. Second, new file systems should be able to call VFS functions. Third, the kernel should export all symbols that may be needed by new loadable kernel modules. The last two requirements are needed only for loadable kernel modules.

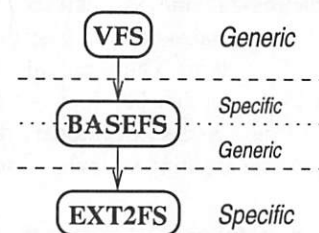


Figure 7: Where Basefs fits inside the kernel

Basefs handles many of the internal details of operating systems, thus freeing developers from dealing with kernel specifics. Basefs provides a stacking layer that is independent from the layers above and below it. Figure 7 shows this. Basefs appears to the upper VFS as a lower level file system. Basefs also appears to file systems below it as a VFS. All the while, Basefs repeats the same vnode operation on the lower level file system.

Basefs performs all data reading and writing on whole pages. This simplifies mixing regular reads and writes with

memory-mapped operations, and gives developers a single paged-based interface to work with. Currently, file systems derived from Basefs manipulate data in whole pages and may not change the data size (e.g., compression).

To improve performance, Basefs copies and caches data pages in its layer and the layers below it.<sup>1</sup> Basefs saves memory by caching at the lower layer only if file data is manipulated and fan-in was used; these are the usual conditions that require caching at each layer.

Basefs is different from Wrapfs in four ways. First, substantial portions of code to manipulate file data and file names, as well as debugging code are not included in Basefs by default. These are included only if the file system needs them. By including only code that is necessary we generate output code that is more readable than code with multi-nested `#ifdef/#endif` pairs. Conditionally including this code also resulted in improved performance, as reported in Section 5.3. Matching or exceeding the performance of other layered file systems was one of the design goals for Basefs.

Second, Basefs adds support for fan-out file systems natively. This code is also conditionally included, because it is more complex than single-stack file systems, adds more performance overhead, and consumes more memory. A complete discussion of the implementation and behavior of fan-out file systems is beyond the scope of this paper.

Third, Basefs includes (conditionally compiled) support for many other features that had to be written by hand in Wrapfs. This added support can be thought of as a library of common functions: opening, reading or writing, and then closing arbitrary files; storing extended attributes persistently; user-level utilities to mount and unmount file systems, as well as manipulate `ioctl`s; inspecting and modifying file attributes, and more.

Fourth, Basefs includes special *tags* that help `fistgen` locate the proper places to insert certain code. Inserting code at the beginning or the end of functions is simple, but in some cases the code to add has to go elsewhere. For example, handling newly defined `ioctl`s is done (in the `basefs_ioctl` vnode function) at the end of a C “switch” statement, right before the “default:” case.

### 3 Implementation

We implemented the FiST system in Solaris, Linux, and FreeBSD because these three operating systems span the most popular modern Unix platforms and they are sufficiently different from each other. This forced us to understand the generic problems in addition to the system-specific problems. Also, we had access to kernel sources for all three platforms, which proved valuable during the

<sup>1</sup>Heidemann proposed a solution to the cache coherency problem through a centralized cache manager[6]. His solution, however, required modifications to existing file systems and the rest of the kernel.

development of our templates. Finally, all three platforms support loadable kernel modules, which sped up the development and debugging process. Loadable kernel modules are a convenience in implementing FiST; they are not required.

The implementation of Basefs was simple and improved on previously reported efforts[27]. No changes were required to either Solaris or FreeBSD. No changes to Linux were required if using statically linked modules. To use dynamically loadable kernel modules under Linux, only three lines of code were changed in a header file. This change was passive and did not have any impact on the Linux kernel.

The remainder of this section describes the implementation of `fistgen`. `Fistgen` translates FiST code into C code which implements the file system described in the FiST input file. The code can be compiled as a dynamically loadable kernel module or statically linked with a kernel. In this section we describe the implementation of key features of FiST that span its full range of capabilities.

We implemented read-only execution environment variables (Section 2.3.1) such as `%uid` by looking for them in one of the fields from `struct cred` in Solaris or `struct ucred` in FreeBSD. The VFS passes these structures to vnode functions. The Linux VFS simplifies access to credentials by reading that information from the disk inode and into the in-memory vnode structure, `struct inode`. So on Linux we find UID and other credentials by referencing a field directly in the inode which the VFS passes to us.

Most of the vnode attributes listed Section 2.3.1 are simple to find. On Linux they are part of the main vnode structure. On Solaris and FreeBSD, however, we first perform a `VOP_GETATTR` vnode operation to find them, and then return the appropriate field from the structure that the `getattr` function fills.

The vnode attribute “name” was more complex to implement, because most kernels do not store file names after the initial name lookup routine translates the name to a vnode. On Linux, implementing the vnode name attribute was simple, because it is part of a standard directory entry structure, `dentry`. On Solaris and FreeBSD, however, we add code to the lookup vnode function that stores the initial file name in the private data of the vnode. That way we can access it as any other vnode attribute, or any other per-vnode attribute added using the `per_vnode` declaration. We implemented all other fields defined using the `per_vfs` FiST declaration in a similar fashion.

The FiST declarations described in Section 2.3 affect the overall behavior of the generated file system. We implemented the read-only access mode by replacing the call part of every file system function that modifies state (such as `unlink` and `mkdir`) to return the error code “read-only file system.” We implemented the fan-in mount style by ex-



cluding code that uses the mounted directory's vnode also as the mount point.

The only difficult part of implementing the `ioctl` declaration and its associated functions, `fistGetIoctlData` and `fistSetIoctlData` (Section 2.2), was finding how to copy data between user space and kernel space. Solaris and FreeBSD use the routines `copyin` and `copyout`; Linux 2.3 uses `copy_from_user` and `copy_to_user`.

The last complex feature we implemented was the fileformat FiST declaration and the functions used with it: `fistGetFileData` and `fistSetFileData` (Section 2.3.1). Consider this small code excerpt:

```
fileformat fmt { data structure; }
fistGetFileData(file, fmt, field, out);
```

First, we generate a C data structure named `fmt`. To implement `fistGetFileData`, we open `file`, read as many bytes from it as the size of the data structure, map these bytes onto a temporary variable of the same data structure type, copy the desired `field` within that data structure into `out`, close the file, and finally return a error/success status value from the function. To improve performance, if fileformat related functions are called several times inside a vnode function, we keep the file they refer to open until the last call that uses it.

Fistgen itself (excluding the templates) is highly portable, and can be compiled on any Unix system. The total number of source lines for fistgen is 4813. Fistgen can process each 1KB of template data in under 0.25 seconds (measured on the same platform used in Section 5.3).

## 4 Examples

This section describes the design and implementation of several sample file systems we wrote using FiST. The examples generally progress from those with a simple FiST design to those with a more complex design. Each example introduces a few more FiST features.

1. **Cryptfs**: is an encryption file system.
2. **Aclfs**: adds simple access control lists.
3. **Unionfs**: joins the contents of two file systems.

These examples are experimental and intended to illustrate the kinds of file systems that can be written using FiST. We illustrate and discuss only the more important parts of these examples—those that depict key features of FiST. Whenever possible, we mention potential enhancements to our examples. We hope to convince readers of the flexibility and simplicity of writing new file systems using FiST. An additional example, Snoopfs, was described in Section 2.1.

### 4.1 Cryptfs

Cryptfs is a strong encryption file system. It uses the Blowfish[21] encryption algorithm in Cipher Feedback (CFB) mode[20]. We used one fixed Initialization Vector (IV) and one 128-bit key per mounted instance of Cryptfs. Cryptfs encrypts both file data and file names. After encrypting file names, Cryptfs also uuencodes them to avoid characters that are illegal in file names. Additional design and important details are available elsewhere[26].

The FiST implementation of Cryptfs shows three additional features: file data encoding, using `ioctl` calls, and using per-VFS data. Cryptfs's FiST code uses all four sections of a FiST file. Some of the more important code for Cryptfs is:

```
%{
#include <blowfish.h>
}%
filter:data;
filter:name;
ioctl:fromuser SETKEY {char ukey[16];};
per_vfs {char key[16];};
%%
%op:ioctl:SETKEY {
    char temp_buf[16];
    if (fistGetIoctlData(SETKEY, ukey, temp_buf)<0)
        fistSetErr(EFAULT);
    else
        BF_set_key(&$vfs.key, 16, temp_buf);
}
%%
unsigned char global_iv[8] = {
    0xfe,0xdc,0xba,0x98,0x76,0x54,0x32,0x10 };
int cryptfs_encode_data(const page_t *in,
                        page_t *out)
{
    int n = 0; /*blowfish variables*/
    unsigned char iv[8];

    fistMemCpy(iv, global_iv, 8);
    BF_cfb64_encrypt(in, out, %pagesize,
        &($vfs.key), iv, &n,
        BF_ENCRYPT);

    return %pagesize;
}
...
```

The above example omits the call to decode data and the calls to encode and decode file names because they are similar in behavior to data encoding. Cryptfs defines an `ioctl` named SETKEY, used to set 128-bit encryption keys. We wrote a simple user-level tool which prompts the user for a passphrase and sends an MD5-hash of it to the kernel using this `ioctl`. When the SETKEY `ioctl` is called, Cryptfs stores the (cipher) key in the private VFS data field "key", to be used later.

There are several possible extensions to Cryptfs: storing per-file or per-directory keys in auxiliary files that would otherwise remain hidden from users' view, much the same as Aclfs does (Section 4.2.); using several types of encryp-



tion algorithms, and defining mount flags to select among them.

## 4.2 Aclfs

Aclfs allows an additional UID and GID to share access to a directory as if they were the owner and group of that directory. Aclfs shows three additional features of FiST: disallowing fan-in (more secure), using special purpose auxiliary files, and hiding files from users' view. The FiST code for Aclfs uses the FiST Declarations and FiST Rules sections:

```
fanin no;
ioctl:fromuser SETACL {int u; int g;};
fileformat ACLDATA {int us; int gr;};
%%
%op:ioctl:SETACL {
    if ($0.owner == %uid) {
        int u2, g2;
        if (fistGetIoctlData(SETACL, u, &u2) < 0 ||
            fistGetIoctlData(SETACL, g, &g2) < 0)
            fistSetErr(EFAULT);
        else {
            fistSetFileData(".acl", ACLDATA, us, u2);
            fistSetFileData(".acl", ACLDATA, gr, g2);
        }
    } else
        fistSetErr(EPERM);
}
%op:lookup:postcall {
    int u2, g2;
    if (fistLastErr() == EPERM
        &&
        fistGetFileData(".acl", ACLDATA, us, u2) >= 0
        &&
        fistGetFileData(".acl", ACLDATA, gr, g2) >= 0
        &&
        (%uid == u2 || %gid == g2))
        fistLookup($dir:1, $name, $1,
            $dir:1.owner, $dir:1.group);
}
%op:lookup:precall {
    if (fistStrEq($name, ".acl") &&
        $dir.owner != %uid)
        fistReturnErr(ENOENT);
}
%op:readdir:call {
    if (fistStrEq($name, ".acl"))
        fistSkipName($name);
}
```

When looking up a file in a directory, Aclfs first performs the normal access checks (in lookup). We insert postcall code after the normal lookup that checks if access to the file was denied and if an additional file named `.acl` exists in that directory. We then read one UID and GID from the `.acl` file. If the effective UID and GID of the current process match those listed in the `.acl` file, we repeat the lookup operation on the originally looked-up file, but using the ownership and group credentials of the *actual* owner of the directory. We must use the owner's credentials, or the lower file system will deny our request.

The `.acl` file itself is modifiable only by the directory's owner. We accomplish this by using a special ioctl. Finally, we hide `.acl` files from anyone but their owner. We insert code in the beginning of lookup that returns the error "no such file" if anyone other than the directory's owner attempted to lookup the ACL file. To complete the hiding of ACL files, we skip listing `.acl` files when reading directories.

Aclfs shows the full set of arguments to the `fistLookup` routine. In order, the five arguments are: the directory to lookup in, the name to lookup, the vnode to store the newly looked up entry, and the credentials to perform the lookup with (UID and GID, respectively).

There are several possible extensions to this implementation of Aclfs. Instead of using the UID and GID listed in the `.acl` file, it can contain an arbitrarily long list of user and group IDs to allow access to. The `.acl` file may also include sets of permissions to deny access from, perhaps using negative integers to distinguish them from access permissions. The granularity of Aclfs can be made on a per-file basis; for each file *F*, access permissions can be read from a file `.F.acl`, if one exists.

## 4.3 Unionfs

Unionfs joins the contents of two file systems similar to the union mounts in BSD-4.4[15] and Plan 9[17]. The two lower file systems can be considered two branches of a stackable file system tree. Unionfs shows how to merge the contents of directories in FiST, and how to define behavior on a set of file system operations. The FiST code for Unionfs uses the FiST Declarations and FiST Rules sections:

```
fanout 2;
%%
%op:lookup:postcall {
    if (fistLastErr() == ENOENT)
        fistSetErr(fistLookup($dir:2, $name));
}
%op:readdir:postcall {
    fistSetErr(fistReaddir($dir:2, NODUPS));
}
%delops:all:postcall {
    fistSetErr(fistOp($2));
}
%writeops:all:call {
    fistSetErr(fistOp($1));
}
```

Normal lookup will try the first lower file system branch (\$1). We add code to lookup in the second branch (\$2) if the first lookup did not find the file. If a file exists in both lower file systems, Unionfs will use the one from the first branch. Normal directory reading is augmented to include the contents of the second branch, but setting a flag to eliminate duplicates; that way files that exist in both lower file systems are listed only once. Since files may exist in

both branches, they must be removed (unlink, rmdir, and rename) from all branches. Finally we declare that all writing operations should perform their respective operations only on the first branch; this means that new files are created in the first branch where they will be found first by subsequent lookups.

There are several other issues file system semantics and especially concerning error propagation and partial failures, but these are beyond the scope of this paper. Extensions to our Unionfs include larger fan-outs, masking the existence of a file in \$2 if it was removed from \$1, and ioctl's or mount options to decide the order of lookups and writing operations on the individual file system branches.

## 5 Evaluation

We evaluate the effectiveness of FiST using three criteria: code size, development time, and performance. We show how code size is reduced dramatically when using FiST, and the corresponding improvements in development and porting times. We also show that performance overhead is small and comparable to other stacking work. We report results based on the four example file systems described in this paper: Snoopfs, Cryptfs, Aclfs, and Unionfs. These were tested on three different platforms: Linux 2.3, Solaris 2.6, and FreeBSD 3.3.

### 5.1 Code Size

Code size is one measure of the development effort necessary for a file system. To demonstrate the savings in code size achieved using FiST, we compare the number of lines of code that need to be written to implement the four example file systems in FiST versus three other implementation approaches: writing C code using a stand-alone version of Basefs, writing C code using Wrapfs, and writing the file systems from scratch as kernel modules using C. In particular, we first wrote all four of the example file systems from scratch before writing them using FiST. For these example file systems, the C code generated from FiST was identical in size (modulo white-spaces and comments) to the hand-written code. We chose to include results for both Basefs and Wrapfs because the latter was released last year, and includes code that makes writing some file systems easier with Wrapfs than Basefs directly.

When counting lines of code, we excluded comments, empty lines, and %% separators. For Cryptfs we excluded 627 lines of C code of the Blowfish encryption algorithm, since we did not write it. When counting lines of code for implementing the example file systems using the Basefs and Wrapfs stackable templates, we exclude code that is part of the templates and only count code that is specific to the given example file system. We then averaged the code

sizes for the three platforms we implemented the file systems on: Linux 2.3, Solaris 2.6, and FreeBSD 3.3. These results are shown in Figure 8. For reference, we include the code sizes of Basefs and Wrapfs and also show the number of lines of code required to implement Wrapfs in FiST and Basefs.

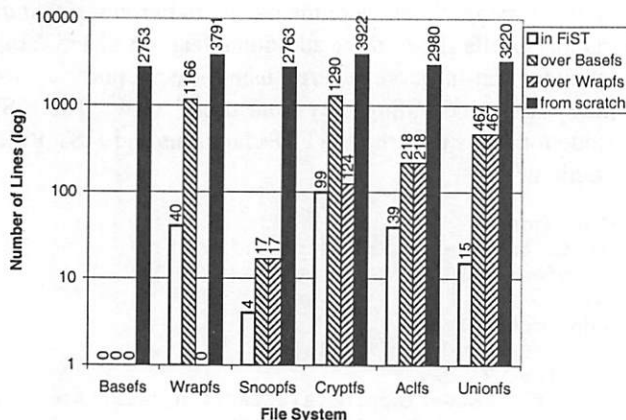


Figure 8: Average code size for various file systems when written in FiST, written given the Basefs or Wrapfs templates, and written from scratch in C.

Figure 8 shows large reductions in code size when comparing FiST versus code hand-written from scratch—generally writing tens of lines instead of thousands. We also include results for the two templates. Size reductions for the four example file systems range from a factor of 40 to 691, with an average of 255. We focus though on the comparison of FiST versus stackable template systems. As Wrapfs represents the most conservative comparison, the figure shows for each file system the additional number of lines of code written using Wrapfs. The smallest average code size reduction in using FiST versus Wrapfs or Basefs across all four file systems ranges from a factor of 1.3 to 31.1; the average reduction rate is 10.5.

Figure 8 suggests two size reduction classes. First, moderate (5–6 times) savings are achieved for Snoopfs, Cryptfs, and Aclfs. The reason for this is that some lines of FiST code for these file systems produce ten or more lines of C code, while others result in almost a one-to-one translation in terms of number of lines.

Second, the largest savings appeared for Unionfs, a factor of 28–33 times. The reason for this is that fan-out file systems produce C code that affects all vnode operations; each vnode operation must handle more than one lower vnode. This additional code was not part of the original Wrapfs implementation, and it is not used unless fan-outs of two or more are defined (to save memory and improve performance). If we exclude the code to handle fan-outs, Unionfs's added C code is still over 100 lines producing savings of a factor of 7–10. FreeBSD's Unionfs is 4863 lines long, which is 50% larger than our Unionfs (3232

lines). FreeBSD's Unionfs is 2221 lines longer than their Nullfs, while ours is only 481 lines longer than our Basefs.<sup>2</sup>

Figure 8 shows the code sizes for *each* platform. The savings gained by FiST are multiplied with each port. If we sum up the savings for the above three platforms, we reach reduction factors ranging from 4 to over 100 times when comparing FiST to code written using the templates. This aggregated reduction factor exceeds 750 times when comparing FiST to C code written from scratch. The more ports of Basefs exist, the better these cumulative savings would be.

## 5.2 Development Time

Estimating the time to develop kernel software is very difficult. Developers' experience can affect this time significantly, and this time is generally reduced with each port. In this section we report our own personal experiences given these file system examples and the three platforms we worked with; these figures do not represent a controlled study. Figure 9 shows the number of days we spent developing various file systems and porting them to three different platforms.

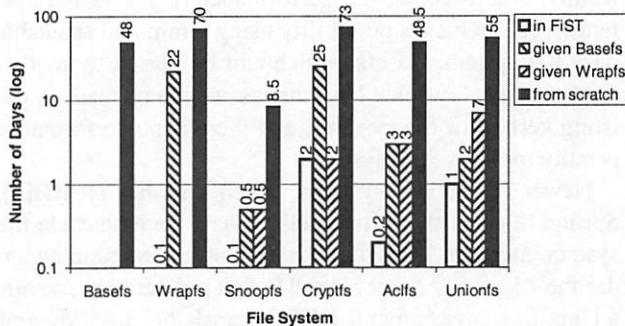


Figure 9: Average estimated reduction in development time

We estimated the incremental time spent designing, developing, and debugging each file system, assuming 8 hour work days, and using our source commit logs and change logs. We estimated the time it took us to develop Wrapfs, Basefs, and the example file systems. Then we measured the time it took us to develop each of these file systems using the FiST language.

For most file systems, incremental time savings are a factor of 5–15 because hand writing C code for each platform can be time consuming, while FiST provides this as part of the base templates and the additional library code that comes with Basefs. For Cryptfs, however, there are no time savings per platform, because the vast majority of the code for Cryptfs is in implementing the four encoding and decoding functions, which are implemented in C code in the

<sup>2</sup>Unfortunately, the stacking infrastructure in FreeBSD is currently broken, so we were unable to compare the performance of our stacking to FreeBSD's.

Additional C Code section of the FiST file; the rest of the support for Cryptfs is already in Wrapfs.

The average per platform reduction in development time across the four file systems is a factor of seven in using FiST versus the Wrapfs templates. If we assume that development time correlates directly to productivity, we can corroborate our results with Brooks's report that high-level languages are responsible for at least a factor of five in improved productivity[3].

An additional metric of productivity is comparing the number of lines of C code developed for each man-day, given the templates. The average number of lines of code we wrote per man-day was 80. One user of our Wrapfs templates had used them to create a new migration file system called mfs<sup>3</sup>. The average number of lines of code he wrote per man-day was 68. The difference between his rate of productivity and ours is only 20%, which can be explained because we are more experienced in writing file systems than he is.

The most obvious savings in development time come when taking into account multiple platforms. Then it is clearer that each additional platform increases the savings factor of FiST versus other methods by one more.

## 5.3 Performance

To evaluate the performance of file systems written using FiST, we tested each of the example file systems by mounting it on top of a disk based native file system and running benchmarks in the mounted file system. We conducted measurements for Linux 2.3, Solaris 2.6, and FreeBSD 3.3. The native file systems used were EXT2, UFS, and FFS, respectively. We measured the performance of our file systems by building a large package: am-utils-6.0, which contains about 50,000 lines of C code in several dozen small files and builds eight binaries; the build process contains a large number of reads, writes, and file lookups, as well as a fair mix of most other file system operations. Each benchmark was run once to warm up the cache for executables, libraries, and header files which are *outside* the tested file system; this result was discarded. Afterwards, we took 10 new measurements and averaged them. In between each test, we unmounted the tested file system and the one below it, and then remounted them; this ensured that we started each test on a cold cache for that file system. The standard deviations for our measurements were less than 2% of the mean. We ran all tests on the same machine: a P5/90, 64MB RAM, and a Quantum Fireball 4.35GB IDE hard disk.

Figure 10 shows the performance overhead of each file system compared to the one it was based on. The intent of these figures is two-fold: (1) to show that the basic stacking overhead is small, and (2) to show the performance benefits

<sup>3</sup><http://www-internal.alphanet.ch/~schaefer/mfs.html>



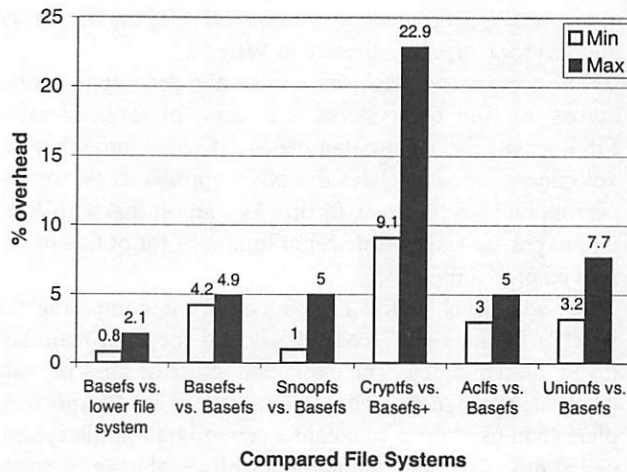


Figure 10: Performance overhead of various file systems for the large compile benchmark, across three operating systems

of conditionally including code for manipulating file names and file data in Basefs. Basefs+ refers to Basefs with code for manipulating file names and file data.

The most important performance metric is the basic overhead imposed by our templates. The overhead of Basefs over the file systems it mounts on is just 0.8–2.1%. This minimum overhead is below the 3–10% degradation previously reported for null-layer stacking[8, 22]. In addition, the overhead of the example file systems due to new file system functionality is greater than the basic stacking overhead imposed by our templates in all cases, even for very simple file systems. With regard to performance, developers who extend file system functionality using FiST primarily need to be concerned with the performance cost of new file system functionality as opposed to the cost of the FiST stacking infrastructure. For instance, the overhead of Cryptfs is the largest of all the file systems shown due to the cost of the Blowfish cipher. Note that the performance of individual file systems can vary greatly depending on the operating system in question.

Figure 10 also shows the benefits of having FiST customize the generated file system infrastructure based on the file system functionality required. The comparison of Basefs+ versus Basefs shows that the overhead of including code for manipulating file names and file data is 4.2–4.9% over Basefs. This added overhead is not incurred in Basefs unless the file systems derived from it requires file data or file name manipulations. While Cryptfs requires Basefs+ functionality, Snoopfs, Aclfs, and Unionfs do not. Compared to a stackable file system such as Wrapfs, FiST's ability to conditionally include file system infrastructure code saves an additional 4% of performance overhead for Snoopfs, Aclfs, and Unionfs.

We also performed several micro-benchmarks which included a series of recursive copies (`cp -r`), recursive re-

movals (`rm -rf`), recursive find, and “find-grep” (`find /mnt -print | xargs grep pattern`) using the same file set used for the large compile. The focus of this paper is not on performance, but on savings in code size and development time. Since the micro-benchmarks confirmed our previous good results, we do not repeat them here[27].

Finally, since we did not change the VFS, and all of our stacking work is in the templates, there is no overhead on the rest of the system; performance of native file systems (NFS, FFS, etc.) is unaffected when our stacking is not used.

## 6 Related Work

Rosenthal first implemented stacking in SunOS 4.1 in the early 1990s[19]. A few other projects followed his, including further prototypes for extensible file systems in SunOS[22], and the Ficus layered file system[5, 7]. Webber implemented file system interface extensions that allow user-level file servers[25]. Unfortunately, these implementations required modifications to either existing file systems or the rest of the kernel, limiting their portability significantly, and affecting the performance of native file systems. FiST achieves portability using a minimal stackable base file system, Basefs, which can be ported to another platform in 1–3 weeks. No changes need to be made to existing kernels or file systems, and there is no performance penalty for native file systems.

Newer operating systems, such as the HURD[4], Spring[13], and the Exokernel[10] have an extensible file system interface. The HURD is a set of servers running under the Mach 3.0 microkernel[1] that collectively provide a Unix-like environment. HURD translators are programs that can be attached to a pathname and perform specialized services when that pathname is accessed. Writing translators entails implementing a well defined file access interface and filling in stub operations for reading files, creating directories, listing directory contents, etc.

Sun Microsystems Laboratories built Spring, an object-oriented research operating system[13]. Spring was designed as a set of cooperating servers on top of a microkernel. It provides generic modules that offer services useful for a file system: caching, coherency, I/O, memory mapping, object naming, and security. Writing a file system for Spring involves defining the operations to be applied on the objects. Operations not defined are inherited from their parent object. One work that resulted from Spring is the Solaris MC (Multi-Computer) File System[12]. It borrowed the object-oriented interfaces from Spring and integrated them with the existing Solaris vnode interface to provide a distributed file system infrastructure through a special *Proxy File System*. Solaris MC provides all of Spring's benefits, while requiring little or no change to existing file systems; those can be ported gradually over time.



Solaris MC was designed to perform well in a closely coupled cluster environment (not a general network) and requires high performance networks and nodes.

The Exokernel is an extensible operating system that comes with XN, a low-level in-kernel stable storage system[10]. XN allows users to describe the on-disk data structures and the methods to implement them (along with file system libraries called libFSes). The Exokernel requires significant porting work to each new platform, but then it can run many unmodified applications.

The main disadvantages of the HURD, Spring, and the Exokernel are that they are not portable enough, not sufficiently developed or stable, or they are not available for general use. In comparison, FiST provides portable stacking on widely available operating systems. Finally, none of the related extensible file systems come with a high-level language that developers can use to describe file systems.

High level languages have seldom been used to generate code for operating system components. FiST is the first major language to describe a large component of the operating system, the file system. Previous work in the area of operating system component languages includes a language to describe video device drivers[24].

## 7 Conclusions

The main contribution of this work is the FiST language which can describe stackable file systems. This is a first time a high-level language has been used to describe stackable file systems. From a single FiST description we generate code for different platforms. We achieved this portability because FiST uses an API that combines common features from several vnode interfaces. FiST saves its developers from dealing with many kernel internals, and lets developers concentrate on the core issues of the file system they are developing. FiST reduces the learning curve involved in writing file systems, by enabling non-experts to write file systems more easily.

The most significant savings FiST offers is in reduced development and porting time. The average time it took us to develop a stackable file system using FiST was about seven times faster than when we wrote the code using Basefs. We showed how FiST descriptions are more concise than hand-written C code: 5–8 times smaller for average stackable file systems, and as much as 33 times smaller for more complex ones. FiST generates file system modules that run in the kernel, thus benefiting from increased performance over user level file servers. The minimum overhead imposed by our stacking infrastructure is 1–2%.

FiST can be ported to other Unix platforms in 1–3 weeks, assuming the developers have access to kernel sources. The benefits of FiST are multiplied each time it is ported to a new platform: existing file systems described with FiST can be used on the new platform without modification.

## 7.1 Future Work

We are developing support for file systems that change sizes such as for compression. The main complexity with supporting compression is that the file offsets at the upper and lower layers are no longer identical, and some form of efficient mapping is needed for operations such as appending to a file or writing in the middle. This code complicates the templates, but makes no change to the language.

We are also exploring layer collapsing in FiST: a method to generate one file system that merges the functionality from several FiST descriptions, thus saving the per-layer stacking overheads.

We plan to port our system to Windows NT. NT has a different file system interface than Unix's vnode interface. NT's I/O subsystem defines its file system interface. NT *Filter Drivers* are optional software modules that can be inserted above or below existing file systems[14]. Their task is to intercept and possibly extend file system functionality. One example of an NT filter driver is its virus signature detector. It is possible to emulate file system stacking under NT. We estimate that porting Basefs to NT will take 2–3 months, not 1–3 weeks as we predict for Unix ports.

## 8 Acknowledgments

We would like to thank the anonymous USENIX reviewers and our shepherd Keith Smith, for their helpful comments in reviewing this paper. This work was partially made possible by NSF infrastructure grants numbers CDA-90-24735 and CDA-96-25374.

## References

- [1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. *USENIX Conf. Proc.*, pages 93–112, Summer 1986.
- [2] B. N. Bershad and C. B. Pinkerton. Watchdogs: Extending the UNIX File System. *USENIX Conf. Proc.*, pages 267–75, Winter 1988.
- [3] F. Brooks. “No Silver Bullet” Refired. In *The Mythical Man-Month, Anniversary Ed.*, pages 205–26. Addison-Wesley, 1995.
- [4] M. I. Bushnell. The HURD: Towards a New Strategy of OS Design. *GNU's Bulletin*. Free Software Foundation, 1994. Copies are available by writing to gnu@prep.ai.mit.edu.
- [5] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page Jr., G. J. Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. *USENIX Conf. Proc.*, pages 63–71, Summer 1990.
- [6] J. Heidemann and G. Popek. Performance of cache coherence in stackable filing. *Fifteenth ACM SOSP*. ACM SIGOPS, 1995.

- [7] J. S. Heidemann and G. J. Popek. A layered approach to file system development. Tech-report CSD-910007. UCLA, 1991.
- [8] J. S. Heidemann and G. J. Popek. File System Development with Stackable Layers. *ACM ToCS*, 12(1):58–89, Feb., 1994.
- [9] S. C. Johnson. *Yacc: Yet Another Compiler-Compiler*, UNIX Programmer's Manual Volume 2 — Supplementary Documents. Bell Laboratories, Murray Hill, New Jersey, July 1978.
- [10] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. *Sixteenth ACM SOSP*, pages 52–65, 1997.
- [11] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. *USENIX Conf. Proc.*, pages 238–47, Summer 1986.
- [12] V. Matena, Y. A. Khalidi, and K. Shirriff. Solaris MC File System Framework. Tech-report TR-96-57. Sun Labs, 1996. <http://www.sunlabs.com/technical-reports/1996/abstract-57.html>.
- [13] J. G. Mitchell, J. J. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. An Overview of the Spring System. *CompCon Conf. Proc.*, 1994.
- [14] R. Nagar. Filter Drivers. In *Windows NT File System Internals: A developer's Guide*, pages 615–67. O'Reilly, 1997.
- [15] J. S. Pendry and M. K. McKusick. Union mounts in 4.4BSD-Lite. *USENIX Conf. Proc. on UNIX and Advanced Computing Systems*, pages 25–33, Winter 1995.
- [16] J. S. Pendry and N. Williams. Amd – The 4.4 BSD Auto-mounter. User Manual, edition 5.3 alpha. March 1991.
- [17] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. *Proceedings of Summer UKUUG Conference*, pages 1–9, July 1990.
- [18] D. S. H. Rosenthal. Requirements for a “Stacking” Vnode/VFS Interface. UI document SD-01-02-N014. UNIX International, 1992.
- [19] D. S. H. Rosenthal. Evolving the Vnode Interface. *USENIX Conf. Proc.*, pages 107–18. USENIX, Summer 1990.
- [20] B. Schneier. Algorithm Types and Modes. In *Applied Cryptography, 2nd ed.*, pages 189–97. John Wiley & Sons, 1996.
- [21] B. Schneier. Blowfish. In *Applied Cryptography, Second Edition*, pages 336–9. John Wiley & Sons, 1996.
- [22] G. C. Skinner and T. K. Wong. “Stacking” Vnodes: A Progress Report. *USENIX Conf. Proc.*, pages 161–74, Summer 1993.
- [23] SMCC. lofs – loopback virtual file system. SunOS 5.5.1 Reference Manual, Section 7. Sun Microsystems, Inc., 20 March 1992.
- [24] S. Thibault, R. Marlet, and C. Consel. A Domain Specific Language for Video Device Drivers: From Design to Implementation. *USENIX Conf. on Domain-Specific Languages*, pages 11–26, 1997.
- [25] N. Webber. Operating System Support for Portable Filesystem Extensions. *USENIX Conf. Proc.*, pages 219–25, Winter 1993.
- [26] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A Stackable Vnode Level Encryption File System. Technical Report CUCS-021-98. Computer Science Department, Columbia University, 1998.
- [27] E. Zadok, I. Badulescu, and A. Shender. Extending File Systems Using Stackable Templates. *USENIX Conf. Proc.*, 1999.

Software, documentation, and additional papers are available from <http://www.cs.columbia.edu/~ezk/research/fist>.

# Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems

Margo I. Seltzer<sup>†</sup>, Gregory R. Ganger<sup>\*</sup>,  
M. Kirk McKusick<sup>‡</sup>, Keith A. Smith<sup>†</sup>, Craig A. N. Soules<sup>\*</sup>, Christopher A. Stein<sup>†</sup>  
<sup>†</sup>*Harvard University*, <sup>\*</sup>*Carnegie Mellon University*, <sup>‡</sup>*Author and Consultant*

## 1 Abstract

The UNIX Fast File System (FFS) is probably the most widely-used file system for performance comparisons. However, such comparisons frequently overlook many of the performance enhancements that have been added over the past decade. In this paper, we explore the two most commonly used approaches for improving the performance of meta-data operations and recovery: journaling and Soft Updates. Journaling systems use an auxiliary log to record meta-data operations and Soft Updates uses ordered writes to ensure meta-data consistency.

The commercial sector has moved en masse to journaling file systems, as evidenced by their presence on nearly every server platform available today: Solaris, AIX, Digital UNIX, HP-UX, Irix, and Windows NT. On all but Solaris, the default file system uses journaling. In the meantime, Soft Updates holds the promise of providing stronger reliability guarantees than journaling, with faster recovery and superior performance in certain boundary cases.

In this paper, we explore the benefits of Soft Updates and journaling, comparing their behavior on both microbenchmarks and workload-based macrobenchmarks. We find that journaling alone is not sufficient to “solve” the meta-data update problem. If synchronous semantics are required (i.e., meta-data operations are durable once the system call returns), then the journaling systems cannot realize their full potential. Only when this synchronicity requirement is relaxed can journaling systems approach the performance of systems like Soft Updates (which also relaxes this requirement). Our asynchronous journaling and Soft Updates systems perform comparably in most cases. While Soft Updates excels in some meta-data intensive microbenchmarks, the macrobenchmark results are more ambiguous. In three cases Soft Updates and journaling are comparable. In a file intensive news workload, journaling prevails, and in a small ISP workload, Soft Updates prevails.

## 2 Introduction

For the past several decades, a recurring theme in operating system research has been file system perfor-

mance. With the large volume of operating systems papers that focus on file systems and their performance, we do not see any change in this trend. Many of the obstacles to high performance file service have been solved in the past decade. For example, clustering of sequential reads and writes removes the need for disk seeks between adjacent files [25][26][28]. The Co-locating FFS [13] solves the inter-file access problem for both reads and writes when the data access pattern matches the namespace locality; that is, when small files in the same directory are accessed together. The synchronous meta-data problem has been addressed most directly through journaling systems [5][16] and Soft Updates systems[11].

In this paper, we focus on the performance impact of synchronous meta-data operations and evaluate the alternative solutions to this problem. In particular, we compare Soft Updates to journaling under a variety of conditions and find that while their performance is comparable, each provides a different set of semantic guarantees.

The contributions of this work are: the design and evaluation of two journaling file systems, both FFS-compatible (i.e., they have the same on-disk structure); a novel journaling architecture where the log is implemented as a stand-alone file system whose services may be used by other file systems or applications apart from the file system; and a quantitative comparison between Soft Updates and journaling.

The rest of this paper is organized as follows. In Section 3, we discuss a variety of techniques for maintaining meta-data integrity. In Section 4, we describe Soft Updates and in Section 5, we discuss our two journaling implementations. In Section 6, we highlight the semantic differences between journaling and Soft Updates. In Section 7, we describe our benchmarking methodology and framework, and in Section 8, we present our experimental results. In Section 9, we discuss related work, and we conclude in Section 10.

## 3 Meta-Data Integrity

File system operations can broadly be divided into two categories, data operations and meta-data operations. Data operations act upon actual user data, reading



or writing data from/to files. Meta-data operations modify the structure of the file system, creating, deleting, or renaming files, directories, or special files (e.g., links, named pipes, etc.).

During a meta-data operation, the system must ensure that data are written to disk in such a way that the file system can be recovered to a consistent state after a system crash. FFS provides this guarantee by requiring that when an operation (e.g., a create) modifies multiple pieces of meta-data, the data must be written to disk in a fixed order. (E.g., a create writes the new inode before writing the directory that references that inode.) Historically, FFS has met this requirement by synchronously writing each block of meta-data. Unfortunately, synchronous writes can significantly impair the ability of a file system to achieve high performance in the presence of meta-data operations. There has been much effort, in both the research community and industry, to remove this performance bottleneck. In the rest of this section, we discuss some of the most common approaches to solving the *meta-data update* problem, beginning with a brief introduction to Soft Updates [11] and journaling [16], the two techniques under analysis here. Then we discuss Soft Updates in detail in Section 4 and journaling in detail in Section 5, and highlight the differences between the two in Section 6.

### 3.1 Soft Updates

Soft Updates attacks the meta-data update problem by guaranteeing that blocks are written to disk in their required order without using synchronous disk I/Os. In general, a Soft Updates system must maintain *dependency information*, or detailed information about the relationship between cached pieces of data. For example, when a file is created, the system must ensure that the new inode reaches disk before the directory that references it does. In order to delay writes, Soft Updates must maintain information that indicates that the directory data block is dependent upon the new inode and therefore, the directory data block cannot be written to disk until after the inode has been written to disk. In practice, this dependency information is maintained on a per-pointer basis instead of a per-block basis in order to reduce the number of cyclic dependencies. This is explained more fully in Section 4.

### 3.2 Journaling Systems

Journaling (or logging) file systems attack the meta-data update problem by maintaining an auxiliary log that records all meta-data operations and ensuring that the log and data buffers are synchronized in such a way to guarantee recoverability. The system enforces

*write-ahead logging* [15], which ensures that the log is written to disk before any pages containing data modified by the corresponding operations. If the system crashes, the log system replays the log to bring the file system to a consistent state. Journaling systems always perform additional I/O to maintain ordering information (i.e., they write the log). However, these additional I/Os can be efficient, because they are sequential. When the same piece of meta-data is updated frequently (e.g., the directory in which a large number of files are being created and that directory's inode), journaling systems incur these log writes in exchange for avoiding multiple meta-data writes.

Journaling systems can provide a range of semantics with respect to atomicity and durability. If the log is maintained synchronously (that is, the log is forced to disk after each meta-data operation), then the journaling system provides guarantees identical to FFS. If the log is maintained asynchronously, buffering log writes until entire buffers are full, the semantics are comparable to Soft Updates.

A third configuration that warrants consideration, but is not currently supported by either of the systems described in this paper, is to support group commit. In a group commit system [16], log writes are synchronous with respect to the application issuing the meta-data operation, but such operations block to allow multiple requests to be batched together, providing the potential for improved log throughput. On a highly concurrent system with many simultaneous meta-data operations, group commit can provide a significant performance improvement, but it provides no assistance on single-threaded applications, such as most of the macrobenchmarks described in Section 7.3. The asynchronous implementations described here provide performance superior to a group commit system since they avoid any synchronous writes and never make applications wait.

In the context of building a journaling file system, the key design issues are:

- Location of the log.
- Management of the log (i.e., space reclamation and checkpointing).
- Integration or interfacing between the log and the main file system.
- Recovering the log.

In Section 5, we present two alternative designs for incorporating journaling in FFS, focusing on how each addresses these issues.

### 3.3 Other Approaches

Some vendors, such as Network Appliance [18], have addressed the meta-data update problem by hard-



ware techniques, most notably non-volatile RAM (NVRAM). Systems equipped with NVRAM not only avoid synchronous meta-data writes, but can cache data indefinitely, safe in the knowledge that data are persistent after a failure. On a system crash, the contents of the NVRAM can be written to disk or simply accessed during the reboot and recovery process. Baker and her colleagues quantify the benefits of such systems[1]. Such systems provide performance superior to both Soft Updates and journaling, but with the additional expense of NVRAM.

The Rio system provides a similar solution [3]. Rio assumes that systems have an uninterrupted power supply, so memory never loses its contents. Part of the normal main memory is treated as a protected region, maintained with read-only protection during normal operation. The region is made writable only briefly to allow updates. This memory is then treated as non-volatile and used during system restart after a crash. Just as with NVRAM, storing meta-data in Rio memory eliminates the need for synchronous writes. The performance trade-off between Rio and Soft Updates or journaling is the cost of protecting and unprotecting the memory region versus maintenance of the dependency or log information.

Log-structured file systems (LFS) offer a different solution to the meta-data update problem. Rather than using a conventional update-in-place file system, log-structured file systems write all modified data (both data blocks and meta-data) in a segmented log. Writes to the log are performed in large segment-sized chunks. By carefully ordering the blocks within a segment, LFS guarantees the ordering properties that must be ensured to update meta-data reliably. Unfortunately, it may not be possible to write all the related meta-data in a single disk transfer. In this case, it is necessary for LFS to make sure it can recover the file system to a consistent state. The original LFS implementation [27] solved this problem by adding small log entries to the beginning of segments, applying a logging approach to the problem. A later implementation of LFS [28] used a simple transaction-like interface to make segments temporary, until all the meta-data necessary to ensure the recoverability of the file system was on disk. LFS utilizes a combination of Soft Updates and journaling approaches. Like Soft Updates, it ensures that blocks are written to disk in a particular order, like journaling, it takes advantage of sequential log writes and log-based recovery.

## 4 Soft Updates

This section provides a brief description of Soft Updates; more detail can be found in other publications [12][14][24].

While conventional FFS uses synchronous writes to ensure proper ordering of meta-data writes, Soft Updates uses *delayed writes* (i.e., write-back caching) for meta-data and maintains dependency information that specifies the order in which data must be written to the disk. Because most meta-data blocks contain many pointers, *cyclic dependencies* occur frequently if dependencies are recorded only at the block level. For example, consider a file creation and file deletion where both file names are in the same directory block and both inodes are in the same inode block. Proper ordering dictates that the newly created inode must be written before the directory block, but the directory block must be written before the deleted inode, thus no single ordering of the two blocks is correct for both cases. In order to eliminate such cyclic dependencies, Soft Updates tracks dependencies on a per-pointer basis instead of a per-block basis. Each block in the system has a list of all the meta-data dependencies that are associated with that block. The system may use any algorithm it wants to select the order in which the blocks are written. When the system selects a block to be written, it allows the Soft Updates code to review the list of dependencies associated with that block. If there are any dependencies that require other blocks to be written before the meta-data in the current block can be written to disk, then those parts of the meta-data in the current block are rolled back to an earlier, safe state. When all needed rollbacks are completed, the initially selected block is written to disk. After the write has completed, the system deletes any dependencies that are fulfilled by the write. It then restores any rolled back values to their current value so that subsequent accesses to the block will see the up-to-date value. These dependency-required rollbacks allow the system to break dependency cycles. With Soft Updates, applications always see the most recent copies of meta-data blocks and the disk always sees copies that are consistent with its other contents.

Soft Updates rollback operations may cause more writes than would be minimally required if integrity were ignored. Specifically, when an update dependency causes a rollback of the contents of an inode or a directory block before a write operation, it must roll the value forward when the write completes. The effect of doing the roll forward immediately makes the block dirty again. If no other changes are made to the block before it is again written to the disk, then the roll forward has generated an extra write operation that would not otherwise have occurred. To minimize the frequency of such extra writes, the syncer task and cache reclamation algorithms attempt to write dirty blocks from the cache in an order that minimizes the number of rollbacks.

Not only does Soft Updates make meta-data write operations asynchronous, it is also able to defer work in

some cases. In particular, when a delete is issued, Soft Updates removes the file's name from the directory hierarchy and creates a remove dependency associated with the buffer holding the corresponding directory data. When that buffer is written, all the delete dependencies associated with the buffer are passed to a separate background syncer task, which does the work of walking the inode and indirect blocks freeing the associated file data blocks. This background deletion typically occurs 30 to 60 seconds after the system call that triggered the file deletion.

If a Soft Updates system crashes, the only inconsistencies that can appear on the disk are blocks and inodes that are marked as allocated when they are actually free. As these are not fatal errors, the Soft Updates file system can be mounted and used immediately, albeit with a possible decrease in the available free space. A background process, similar to `fsck`, can scan the file system to correct these errors [24].

As discussed in more detail in Section 6, while Soft Updates preserves the integrity of the file system, it does not guarantee (as FFS does) that all meta-data operations are durable upon completion of the system call.

## 5 Journaling

In this section, we describe two different implementations of journaling applied to the fast file system. The first implementation (LFFS-file) maintains a circular log in a file on the FFS, in which it records journaling information. The buffer manager enforces a write-ahead logging protocol to ensure proper synchronization between normal file data and the log.

The second implementation (LFFS-wafs) records log records in a separate stand-alone service, a write-ahead file system (WAFS). This stand-alone logging service can be used by other clients, such as a database management system [30], as was done in the Quicksilver operating system [17].

### 5.1 LFFS-file

The LFFS-file architecture is most similar to the commercially available journaling systems. LFFS-file augments FFS with support for write-ahead logging by adding logging calls to meta-data operations. The log is stored in a pre-allocated file that is maintained as a circular buffer and is about 1% of the file system size. To track dependencies between log entries and file system blocks, each cached block's buffer header identifies the first and last log entries that describe an update to the corresponding block. The former is used to ensure that log space is reclaimed only when it is no longer needed, and the latter is used to ensure that all relevant log

entries are written to disk before the block. These two requirements are explained further below.

The fundamental requirement of write-ahead logging is that the logged description of an update must propagate to persistent storage before the updated blocks. The function LFFS-file calls during initiation of disk writes enforces this requirement. By examining the buffer headers of the blocks it is writing, LFFS-file can determine those portions of the log that must first be written. As the log is cached in memory, it must be flushed to disk up to and including the last log entry recorded for the block that is about to be written. In most cases, the relevant log entries will already be on disk, however if they are not, then a synchronous log write is initiated before the block is written. This synchronous flush requires an update of the file system's superblock.

Since the log is implemented as a circular buffer, log space must be reclaimed. LFFS-file uses standard database checkpointing techniques [15]. Specifically, space is reclaimed in two ways. First, during the periodic syncer daemon activity (once per second), the logging code examines the buffer headers of all cached blocks to determine the oldest log entry to which a dirty buffer refers. This becomes the new start of the log, releasing previously live space in the log. The log's start is recorded in the superblock, so that roll-forward can occur efficiently during crash recovery. While this approach is usually sufficient to keep the log from becoming full, the logging code will force a checkpoint when necessary. Such a forced checkpoint causes all blocks with updates described by some range of log entries to be immediately written to persistent storage.

LFFS-file maintains its log asynchronously, so like Soft Updates, it maintains file system integrity, but does not guarantee durability.

LFFS-file is a minor modification to FFS. It requires approximately 35 hooks to logging calls and adds a single new source file of approximately 1,700 lines of code to implement these logging calls.

### 5.2 LFFS-wafs

LFFS-wafs implements its log in an auxiliary file system that is associated with the FFS. The logging file system (WAFS, for Write-Ahead File System) is a simple, free-standing file system that supports a limited number of operations: it can be mounted and unmounted, it can append data, and it can return data by sequential or keyed reads. The keys for keyed reads are log-sequence-numbers (LSNs), which correspond to logical offsets in the log. Like the logging file in LFFS-file, the log is implemented as a circular buffer within the physical space allocated to the file system. When



data are appended to the log, WAFS returns the logical offset at which the data are written. This LSN is then used to tag the data described by the logging operation exactly as is done in LFFS-file (low and high LSNs are maintained for each modified buffer in the cache).

LFFS-wafs uses the same checkpointing scheme as that used for LFFS-file. LFFS-wafs also enforces the standard write-ahead logging protocol as described for LFFS-file.

Because LFFS-wafs is implemented as two disjoint file systems, it provides a great deal of flexibility in file system configuration. First, the logging system can be used to augment any file system, not just an FFS. Second, the log can be parameterized and configured to adjust the performance of the system. In the simplest case, the log can be located on the same drive as the file system. As is the case for LFFS-file, this will necessarily introduce some disk contention between log writes and foreground file system activity. A higher performing alternative is to mount the log on a separate disk, ideally a small, high speed one. In this case, the log disk should never seek and the data disk will perform no more seeks than does a conventional FFS. Finally, the log could be located in a small area of battery-backed-up or non-volatile RAM. This option provides the greatest performance, at somewhat higher cost.

By default, LFFS-wafs mounts its log synchronously so that meta-data operations are persistent upon return from the system call. That is, log messages for creates, deletes, and renames are flushed to disk before the system call returns, while log messages corresponding to bitmap operations are cached in memory until the current log block is flushed to disk. This configuration provides semantics identical to those provided by FFS. For higher performance, the log can be mounted to run asynchronously. In this case, the system maintains the integrity of the file system, but does not provide synchronous FFS durability guarantees; instead it provides semantics comparable to LFFS-file and Soft Updates.

LFFS-wafs requires minimal changes to FFS and to the rest of the operating system. The FreeBSD 4.0 operating system was augmented to support LFFS-wafs by adding approximately 16 logging calls to the ufs layer (that is the Unix file system layer, independent of the underlying file system implementation) and 13 logging calls to manage bitmap allocation in the FFS-specific portion of the code. These logging calls are writes into the WAFS file system. The only other change is in the buffer management code, which is enhanced to maintain and use the LSNs to support write-ahead logging. The buffer management changes required approximately 200 lines of additional code.

Although similar in design to LFFS-file, LFFS-wafs is somewhat more complex. Rather than simply

logging to a file, LFFS-wafs implements the infrastructure necessary to support a file system. This results in about three times the number of lines of code (1,700 versus 5,300).

### 5.3 Recovery

Both journaling file systems require database-like recovery after system failure. First, the log is recovered. In both LFFS-file and LFFS-wafs, a superblock contains a reference to the last log checkpoint. In LFFS-file, the superblock referenced is that of the FFS; in LFFS-wafs, the superblock is that of the WAFS. In LFFS-file, checkpoints are taken frequently and the state described in the superblock is taken as the starting state, thus any log writes that occurred after the last checkpoint are lost. In LFFS-wafs, superblocks are written infrequently and the log recovery code must find the end of the log. It does so by reading the log beginning with the last checkpoint and reading sequentially until it locates the end of the log. Log entries are timestamped and checksummed so that the log recovery daemon can easily detect when the end of the log is reached.

Once the log has been recovered, recovery of the main file system begins. This process is identical to standard database recovery [15]. First, the log is read from its logical end back to the most recent checkpoint and any aborted operations are undone. LFFS-file uses multiple log records for a single meta-data operation, so it is possible that only a subset of those records have reached the persistent log. While database systems typically use a commit record to identify the end of a transaction, LFFS-file uses uniquely identified record types to indicate the end of a logical operation. When such records do not occur, the preceding operations are treated as aborted operations. Since LFFS-wafs logs at a somewhat higher logical level, creates are the only potentially aborted operations. Creates require two log records, one to log the allocation of the inode and one to log the rest of the create. In both LFFS-file and LFFS-wafs, aborted operations must be undone rather than rolled forward. This happens on the backward pass through the log. On the forward pass through the log, any updates that have not yet been written to disk are reapplied. Most of the log operations are idempotent, so they can be redone regardless of whether the update has already been written to disk. Those operations that are not idempotent affect data structures (e.g., inodes) that have been augmented with LSNs. During recovery, the recovery daemon compares the LSN in the current log record to that of the data structure and applies the update only if the LSN of the data structure matches the LSN logged in the record.

	File System Configurations
FFS	Standard FFS
FFS-async	FFS mounted with the async option
Soft-Updates	FFS mounted with Soft Updates
LFFS-file	FFS augmented with a file log log writes are asynchronous
LFFS-wafs-1sync	FFS augmented with a WAFS log log writes are synchronous
LFFS-wafs-1async	FFS augmented with a WAFS log log writes are asynchronous
LFFS-wafs-2sync	FFS augmented with a WAFS log log is on separate disk log writes are synchronous
LFFS-wafs-2async	FFS augmented with a WAFS log log is on a separate disk log writes are asynchronous

Table 1. File System Configurations.

Feature	File Systems
Meta-data updates are synchronous	FFS, LFFS-wafs-[12]sync
Meta-data updates are asynchronous	Soft Updates LFFS-file LFFS-wafs-[12]async
Meta-data updates are atomic.	LFFS-file LFFS-wafs-[12]*
File data blocks are freed in background	Soft Updates
New data blocks are written before inodes	Soft Updates
Recovery requires full file system scan	FFS
Recovery requires log replay	LFFS.*
Recovery is non-deterministic and may be impossible	FFS-async

Table 2. Feature Comparison.

Once the recovery daemon has completed both its backward and forward passes, all the dirty data blocks are written to disk, the file system is checkpointed and normal processing continues. The length of time for recovery is proportional to the inter-checkpoint interval.

## 6 System Comparison

When interpreting the performance results of Section 8, it is important to understand the different systems, the guarantees that they make, and how those guarantees affect their performance. Table 1 lists the different file systems that we will be examining and Table 2 summarizes the key differences between them.

FFS-async is an FFS file system mounted with the async option. In this configuration, all file system writes

are performed asynchronously. Because it does not include the overhead of either synchronous meta-data updates, update ordering, or journaling, we expect this case to represent the best case performance. However, it is important to note that such a file system is not practical in production use as it may be unrecoverable after system failure.

Both journaling and Soft Updates systems ensure the integrity of meta-data operations, but they provide slightly different semantics. The four areas of difference are the durability of meta-data operations such as create and delete, the status of the file system after a reboot and recovery, the guarantees made about the data in files after recovery, and the ability to provide atomicity.

The original FFS implemented meta-data operations such as create, delete, and rename synchronously, guaranteeing that when the system call returned, the meta-data changes were persistent. Some FFS variants (e.g., Solaris) made deletes asynchronous and other variants (e.g., SVR4) made create and rename asynchronous. However, on FreeBSD, FFS does guarantee that create, delete, and rename operations are synchronous.

FFS-async makes no such guarantees, and furthermore does not guarantee that the resulting file system can be recovered (via `fsck`) to a consistent state after failure. Thus, instead of being a viable candidate for a production file system, FFS-async provides an upper bound on the performance one can expect to achieve with the FFS derivatives.

Soft Updates provides looser guarantees than FFS about when meta-data changes reach disk. Create, delete, and rename operations typically reach disk within 45 seconds of the corresponding system call, but can be delayed up to 90 seconds in certain boundary cases (a newly created file in a hierarchy of newly created directories). Soft Updates also guarantees that the file system can be restarted without any file system recovery. At such a time, file system integrity is assured, but freed blocks and inodes may not yet be marked as free and, as such, the file system may report less than the actual amount of free space. A background process, similar to `fsck`, restores the file system to an accurate state with respect to free blocks and inodes [24].

The journaling file systems provide a spectrum of points between the synchronous guarantees of FFS and the relaxed guarantees of Soft Updates. When the log is maintained synchronously, the journaling systems provide guarantees identical to FFS; when the log is written asynchronously, the journaling systems provide guarantees identical to Soft Updates, except that they require a short recovery phase after system restart to make sure that all operations in the log have been applied to the file system.



The third area of different semantics is in the guarantees made about the status of data in recently created or written to files. In an ideal system, one would never allow meta-data to be written to disk before the data referenced by that meta-data are on the disk. For example, if block 100 were allocated to file 1, you would want block 100 to be on disk before file 1's inode was written, so that file 1 was not left containing bad (or highly sensitive) data. FFS has never made such guarantees. However, Soft Updates uses its dependency information to roll back any meta-data operations for which the corresponding data blocks have not yet been written to disk. This guarantees that no meta-data ever points to bad data. In our tests, the penalty for enforcing this ranges from 0 (in the less meta-data intensive `ssh` benchmark described in Section 7.3.1) to approximately 8% (in the meta-data intensive Netnews benchmark, described in Section 7.3.2). Neither of the journaling file systems provides this stronger guarantee. These differences should be taken into account when comparing performance results.

The final difference between journaling systems and Soft Updates is the ability to provide atomicity of updates. Since a journaling system records a logical operation, such as rename, it will always recover to either the pre-operation or post-operation state. Soft Updates can recover to a state where both old and new names persist after a crash.

## 7 Measurement Methodology

The goal of our evaluation is twofold. First, we seek to understand the trade-offs between the two different approaches to improving the performance of meta-data operations and recovery. Second, we want to understand how important the meta-data update problem is to some typical workloads.

We begin with a set of microbenchmarks that quantify the performance of the most frequently used meta-data operations and that validate that the performance difference between the two systems is limited to meta-data operations (i.e., that normal data read and write operations behave comparably). Next, we examine macrobenchmarks.

### 7.1 The Systems Under Test

We compared the two LFFS implementations to FFS, FFS-async, and Soft Updates. Our test configuration is shown in Table 3.

### 7.2 The Microbenchmarks

Our microbenchmark suite is reminiscent of any number of the microbenchmark tests that appear in the

	FreeBSD Platform
Motherboard	Intel ASUS P38F, 440BX Chipset
Processor	500 Mhz Xeon Pentium III
Memory	512 MB, 10 ns
Disk	3 9 GB 10,000 RPM Seagate Cheetahs Disk 1: Operating system, /usr, and swap Disk 2: 9,088 MB test partition Disk 2: 128 MB log partition Disk 3: 128 MB log partition
I/O Adapter	Adaptec AHA-2940UW SCSI
OS	FreeBSD-current (as of 1/26/00 10:30 PM) config: GENERIC + SOFTUPDATES - bpfiler - unnecessary devices

**Table 3. System Configuration.**

file system literature [11][27][29]. The basic structure is that for each of a large number of file sizes, we create, read, write, and delete either 128 MB of data or 512 files, whichever generates the most files. The files are allocated 50 per directory to avoid excessively long lookup times. The files are always accessed in the same order.

We add one microbenchmark to the suite normally presented: a create/delete benchmark that isolates the cost of meta-data operations in the absence of any data writing. The create/delete benchmark creates and immediately deletes 50,000 0-length files, with each newly-created file deleted before moving on to the next. This stresses the performance of temporary file creation/deletion.

The results of all the microbenchmarks are presented and discussed in Section 8.1.

### 7.3 The Macrobenchmarks

The goal of our macrobenchmarking activity is to demonstrate the impact of meta-data operations for several common workloads. As there are an infinite number of workloads, it is not possible to accurately characterize how these systems will benefit all workloads. Instead, we show a variety of workloads that demonstrate a range of effects that meta-data operations can introduce.

#### 7.3.1 The SSH Benchmark

The most widely used benchmark in the file system literature is the Andrew File System Benchmark [19]. Unfortunately, this benchmark no longer stresses the file system, because its data set is too small. We have constructed a benchmark reminiscent of Andrew that does stress a file system.

Our benchmark unpacks, configures, and builds a medium-sized software package (`ssh` version 1.2.26 [34]). In addition to the end-to-end timing measurement, we also measure the time for each of the three phases of this benchmark:

- *Unpack* This phase unpacks a compressed tar archive containing the `ssh` source tree. This phase highlights meta-data operations, but unlike our microbenchmarks, does so in the context of a real workload. (I.e., it uses a mix of file sizes.)
- *Config* This phase determines what features and libraries are available on the host operating system and generates a Makefile reflecting this information. To do this, it compiles and executes many small test programs. This phase should not be as meta-data intensive as the first, but because most of the operations are on small files, there are more meta-data operations than we see in the final phase.
- *Build* This phase executes the Makefile built during the config phase to build the `ssh` executable. It is the most compute-intensive phase of the benchmark (90% CPU utilization running on FFS). As a result, we expect to see the least performance difference here.

We run the three phases of the benchmark consecutively, so the config and build phases run with the file system cache warmed by the previous phases.

### 7.3.2 Netnews

A second workload that we examine is that of a Netnews server. We use a simplified version of Karl Swartz's Netnews benchmark [31]. It simulates the work associated with unbatching incoming news articles and expiring old articles by replaying traces from a live news server. The benchmark runs on a file system that is initialized to contain 2 GB of simulated news data. This data is broken into approximately 520,000 files spread over almost 7,000 directories. The benchmark itself consists of two phases:

- *Unbatch* This phase creates 78,000 new files containing 270 MB of total data.
- *Expire* This phase removes 91,000 files, containing a total of 250 MB of data.

In addition to the sheer volume of file system traffic that this benchmark generates, this workload has two other characteristics that effect the file system. First, successive create and delete operations seldom occur in the same directory. Because FFS places different directories in different regions of the disk, this results in little locality of reference between successive (synchronous) meta-data operations, causing a large number of disk seeks.

The second characteristic of interest is that due to the large data set that the benchmark uses, it is difficult for the file system to maintain all of the meta-data in its buffer cache. As a result, even the Soft Updates and journaling file systems that we are studying may incur many seeks, since the meta-data on which they need to operate may not be in cache. It is important to note that our benchmark is actually quite small compared to current netnews loads. Two years ago, a full news feed could exceed 2.5 GB of data, or 750,000 articles per day [4][7]. Anecdotal evidence suggests that a full news feed today is 15–20 GB per day.

### 7.3.3 SDET

Our third workload is the deprecated SDET benchmark from SPEC. This benchmark was originally designed to emulate a typical timesharing workload, and was deprecated as the computing landscape shifted from being dominated by timesharing systems to being dominated by networked clients and servers [10]. Nonetheless, as SDET concurrently executes one or more scripts of user commands designed to emulate a typical software-development environment (e.g., editing, compiling, and various UNIX utilities), it makes fairly extensive use of the file system. The scripts are generated from a predetermined mix of commands [8][9], and the reported metric is scripts/hour as a function of the script concurrency.

### 7.3.4 PostMark

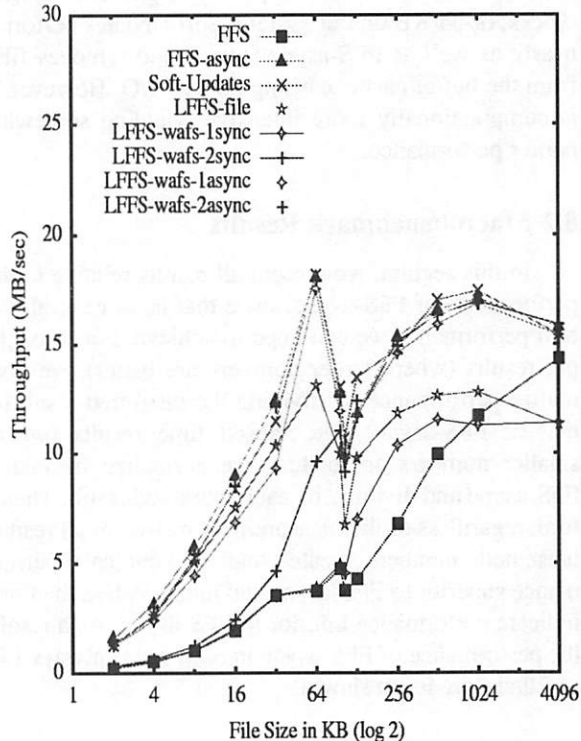
The PostMark benchmark was designed by Jeffrey Katcher to model the workload seen by Internet Service Providers under heavy load [21]. Specifically, the workload is meant to model a combination of electronic mail, netnews, and web-based commerce transactions. To accomplish this, PostMark creates a large set of files with random sizes within a set range. The files are then subjected to a number of transactions. These transactions consist of a pairing of file creation or deletion with file read or append. Each pair of transactions is chosen randomly and can be biased via parameter settings. The file creation operation creates a new file. The sizes of these files are chosen at random and are uniformly distributed over the file size range. File deletion removes a file from the active set. File read selects a random file and reads it in its entirety. File append opens a random file, seeks to the end of the file, and writes a random amount of data, not exceeding the maximum file size. We initially ran our experiments using the default PostMark configuration of 10,000 files with a size range of 512 bytes to 16 KB. One run of this default configuration performs 20,000 transactions with no bias toward any particular transaction type and with a transaction block size of 512

bytes. However, as this workload is far smaller than the workload observed at any ISP today, we ran a larger benchmark using 150,000 files with the default size range, for a total data size of approximately 1.1 GB. The results presented in Section 8.2.4 show both workloads, and it is important to note that the results change dramatically with the data set size. When we increase the data set by a factor of 15, performance (in transactions per second) dropped by nearly the same factor.

## 8 Results

### 8.1 Microbenchmark Results

Our collection of microbenchmarks separates meta-data operations from reading and writing. As the systems under test all use the same algorithms and underlying disk representation, we expect to see no significant performance difference for read and write tests. For the create and delete tests, we expect both Soft Updates and the journaling systems to provide significantly improved performance over FFS. The important question is how close these systems come to approaching the performance of FFS-async, which might be viewed as the best performance possible under any FFS-based system.



**Figure 1. Create Performance as a Function of File Size.**

All of the microbenchmarks represent the average of at least five runs; standard deviations were less than 1% of the average. The benchmarks were run with a cold file system cache.

The read and write tests perform comparably, as expected (and are omitted for the sake of space).

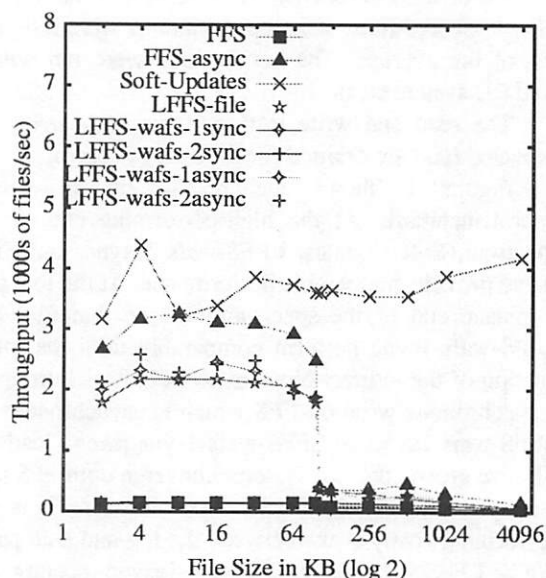
Figure 1 shows the results of the create microbenchmark. At the high-performing end of the spectrum, Soft Updates, LFFS-wafs-2async, and FFS-async provide comparable performance. At the low performance end of the spectrum, we see that FFS and LFFS-wafs-1sync perform comparably until the introduction of the indirect block at 104KB. This introduces a synchronous write on FFS which is asynchronous on LFFS-wafs-1sync, so LFFS-wafs-1sync takes a lead. As file size grows, the two systems converge until FFS ultimately overtakes LFFS-wafs-1sync, because it is not performing costly seeks between the log and data partitions. LFFS-file and LFFS-wafs-1async occupy the region in the middle, reaping the benefits of asynchronous meta-data operations, but paying the penalty of seeks between the data and the log.

The next significant observation is the shape of the curves with the various drops observed in nearly all the systems. These are idiosyncrasies of the FFS disk layout and writing behavior. In particular, on our configuration, I/O is clustered into 64 KB units before being written to disk. This means that at 64KB, many of the asynchronous systems achieve nearly the maximum throughput possible. At 96 KB, we see a drop because we are doing two physically contiguous writes and losing a disk rotation between them. At 104 KB we see an additional drop due to the first indirect block, which ultimately causes an additional I/O. From 104 KB to 1 MB we see a steady increase back up to the maximum throughput. Between 1 and 4 MB there is a slight decline caused by longer seeks between the first 96 KB of a file and the remainder of the file as the larger files fill cylinder groups more quickly.

For small file sizes, where meta-data operations dominate, LFFS-wafs-2async offers a significant improvement over LFFS-wafs-2sync. As file size grows, the benchmark time is dominated by data transfer time and the synchronous and asynchronous systems converge.

The delete microbenchmark performance is shown in Figure 2. Note that performance is expressed in files per second. This microbenchmark highlights a feature of Soft Updates that is frequently overlooked. As explained in Section 4, Soft Updates performs deletes in the background. As a result, the apparent time to remove a file is short, leading to the outstanding performance of Soft Updates on the delete microbenchmark. This backgrounding of deletes provides a very real advantage in





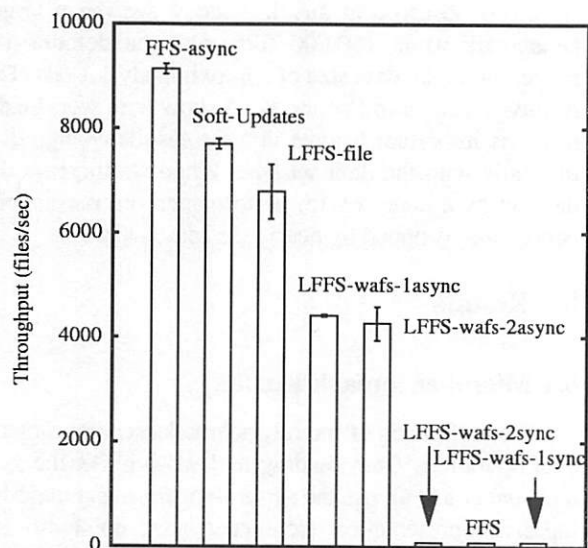
**Figure 2. Delete Performance as a Function of File Size.**

certain workloads (e.g., removing an entire directory tree), while in other cases, it simply defers work (e.g., the Netnews benchmark discussed in Section 7.3).

As soon as the file size surpasses 96 KB, all of the systems without Soft Updates suffer a significant performance penalty, because they are forced to read the first indirect block in order to reclaim the disk blocks it references. In contrast, by backgrounding the delete, Soft Updates removes this read from the measurement path.

In the region up to and including 96 KB, Soft Updates still enjoys increased performance because it performs deletes in the background, but the effect is not as noticeable. The journaling systems write a log message per freed block, so they suffer from a slight decrease in performance as the number of blocks in the file increases.

Our final microbenchmark is the 0-length file create/delete benchmark. This benchmark emphasizes the benefits of asynchronous meta-data operations without the interference of data reads or writes. This benchmark also eliminates the overhead of compulsory read misses in the file system cache, as the test repeatedly accesses the same directory and inode data. Figure 3 shows the results for this benchmark. As this benchmark does nothing outside of meta-data operations, the synchronous journaling implementations behave identically to FFS. The one- and two-disk WAFS-based asynchronous journaling implementations perform comparably, achieving less than half the performance of FFS-async. The reason for FFS-async's superiority is that when the system is running completely asynchronously, the files are created and deleted entirely within the buffer cache



**Figure 3. 0-length File Create/Delete Results in Files per Second.**

and no disk I/O is needed. The journaling systems, however, still write log records. LFFS-file outperforms the WAFS-based journaling schemes because it writes log blocks in larger clusters. The WAFS-based systems write 4 KB log blocks while the LFFS-file system writes in fully clustered I/Os, typically eight file system blocks, or 64 KB on our system. Soft Updates performs nearly as well as FFS-async since it too removes files from the buffer cache, causing no disk I/O. However, it is computationally more intensive, yielding somewhat poorer performance.

## 8.2 Macrobenchmark Results

In this section, we present all results relative to the performance of FFS-async, since that is, in general, the best performance we can hope to achieve. For throughput results (where larger numbers are better), we normalize performance by dividing the measured result by that of FFS-async. For elapsed time results (where smaller numbers are better), we normalize by taking FFS-async and dividing by each measured result. Therefore, regardless of the measurement metric, in all results presented, numbers greater than one indicate performance superior to FFS-async and numbers less than one indicate performance inferior to FFS-async. As a result, the performance of FFS-async in each test is always 1.0, and therefore is not shown.

### 8.2.1 Ssh

As explained in Section 7.3.1, this benchmark simulates unpacking, configuring and building ssh [34].



	Unpack	Config	Build	Total
	Absolute Time (in seconds)			
FFS-async	1.02	10.38	42.60	53.99
	Performance Relative to FFS-async			
FFS	0.14	0.66	0.85	0.73
Soft-Updates	0.99	0.98	1.01	1.01
LFFS-file	0.72	1.08	0.95	0.96
LFFS-wafs-1sync	0.15	1.01	0.88	0.82
LFFS-wafs-1async	0.90	0.94	1.00	0.99
LFFS-wafs-2sync	0.20	0.85	0.93	0.86
LFFS-wafs-2async	0.90	1.05	0.98	0.99

**Table 4. Ssh Results.** Data gathered are the averages of 5 runs; the total column is the measured end-to-end running time of the benchmark. Since the test is not divided evenly into the three phases, the normalized results of the first three columns do not average to the normalized result of the total column. All standard deviations were small relative to the averages. As the config and build phases are the most CPU-intensive, they show the smallest difference in execution time for all systems. Unpack, the most meta-data intensive, demonstrates the most significant differences.

Table 4 reports the normalized performance of our systems. While many of the results are as expected, there are several important points to note. The config and build phases are CPU-intensive, while the unpack phase is dominated by disk-intensive meta-data operations. For the CPU-intensive phases, most of the journaling and Soft Updates systems perform almost as well as FFS-async, with the synchronous journaling systems exhibiting somewhat reduced throughput, due to the few synchronous file creations that must happen.

During the unpack phase, Soft Updates is the only system able to achieve performance comparable to FFS-async. The synchronous journaling systems demonstrate only 10 to 20% improvement over FFS, indicating that the ratio of meta-data operations to data operations is significant and that the meta-data operations account for nearly all the time during this phase. Both the LFFS-wafs-async systems approach 90% of the performance of FFS-async.

The LFFS-file system has slower file create performance on files larger than 64KB, and the build benchmark contains a sufficient number of these to explain its reduced performance on the unpack phase.

### 8.2.2 Netnews

As described in Section 7.3.2, the Netnews benchmark places a tremendous load on the file system, both in terms of the number of meta-data operations it performs, and the amount of data on which it operates. The impact of these stresses is apparent in the benchmark

	Unbatch	Expire	Total
	Absolute Time (in seconds)		
FFS-async	1282	640	1922
	Perf. Relative to FFS-async		
FFS	0.63	0.40	0.53
Soft-Updates	0.86	0.89	0.87
LFFS-file	0.95	0.95	0.95
LFFS-wafs-1sync	0.67	0.48	0.59
LFFS-wafs-1async	0.98	0.92	0.96
LFFS-wafs-2sync	0.91	0.67	0.81
LFFS-wafs-2async	0.97	0.95	0.96

**Table 5. Netnews Results Normalized to FFS-async.** These results are based on a single run, but we observed little variation between multiple runs of any configuration.

results shown in Table 5. On this benchmark, all of the file systems are completely disk bound.

All of the asynchronous journaling systems and the two-disk synchronous system approach the performance of FFS-async (within 5%), but Soft Updates performs at only 87% of FFS-async and the one disk synchronous system performs at less than 60% of FFS-async. The Soft Updates performance is largely due to writes caused by dependency-required rollback. Soft Updates performed 13% more disk writes than FFS. The major cause for these rollbacks is that the data set exceeds the size of the buffer cache. Much of the performance benefits of Soft Updates come from being able to aggregate several meta-data writes into a single write. For example, updating several inodes in a single block at once rather than writing each one individually. To be most effective, it needs to be able to cache blocks for at least 15 and preferably 30 seconds. In the Netnews benchmark, the cache evictions occur much more rapidly, which decreases the aggregation and increases the likelihood of needing to do rollback operations. Recent tuning work (reflected in these results) defers the writing of blocks with rollback dependencies by having them travel around the LRU list twice before they are written. This change eliminated most of the rollbacks associated with directory dependencies. About 65% of the remaining extra I/O operations come from the rollbacks associated with ensuring that inodes not reference data blocks that have not been written (see Section 6 for a discussion of this feature). The other 35% comes from the reduced aggregation caused by the faster buffer flushing and rollbacks associated with directories.

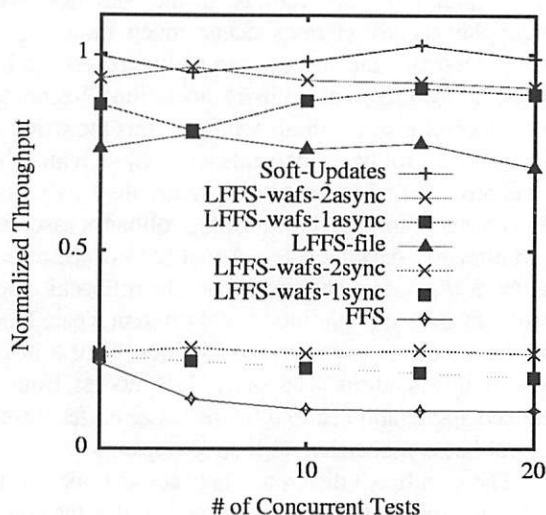
The significant difference between the one and two disk synchronous systems indicates that it is the contention between log I/O and data I/O that hampers performance, not the synchronous writing of the log.

### 8.2.3 SDET

Figure 4 shows the results for the SDET test. Once again we see the systems diverge into the largely asynchronous ones (Soft Updates, LFFS-file, LFFS-wafs-[12]async) and the synchronous ones (FFS, LFFS-wafs-[12]sync), with the synchronous journaling systems providing minimal improvement over FFS. As expected, the synchronous schemes drop in performance as script concurrency increases, because the scripts compete for the disk. Soft Updates outperforms the other schemes because of its backgrounding of file deletion. LFFS-file suffers the same performance problem here that we observed in the `ssh` unpack test, namely that it creates files larger than 64 KB more slowly than the other systems.

### 8.2.4 PostMark

This test, whose results are shown in Figure 5, demonstrates the impact that delayed deletes can have on subsequent file system performance. When we run the test with a small file set (right-hand bars), Soft Updates outperforms the LFFS-wafs systems significantly and outperforms LFFS-file by a small margin. However, with a larger data set (left-hand bars), which takes significantly longer to run (1572 seconds versus 21 seconds for the FFS-async case), the backgrounded deletes interfere with other file system operations and Soft Updates performance is comparable to all the asynchronous journaling systems. When the log writes are synchronous, seeks between the logging and data portions of the disk cause the difference between the 1-disk and 2-disk cases. In the asynchronous case, the ability to write log records lazily removes the disk seeks from the critical path.



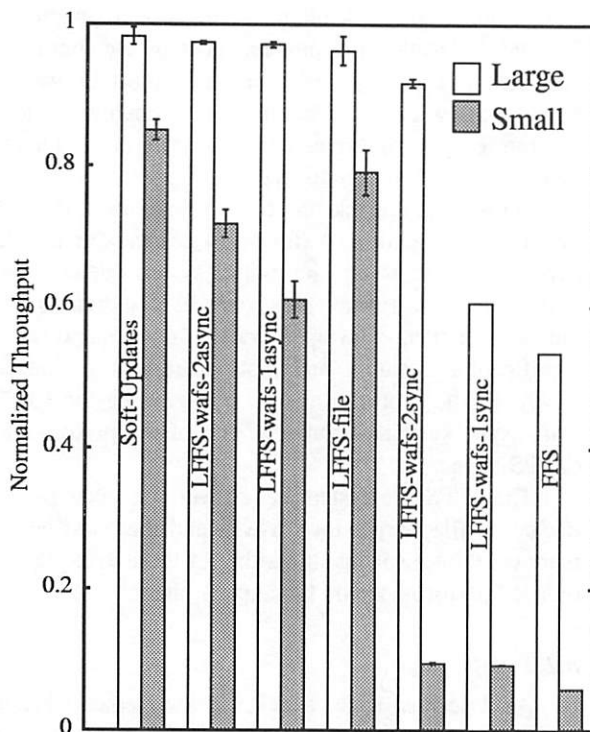
**Figure 4. SDET Results.** The results are the averages of five runs.

## 9 Related Work

In Section 3, we discussed much of the work that has been done to avoid synchronous writes in FFS. As mentioned in the introduction, small writes are another performance bottleneck in FFS. Log-structured file systems [27] are one approach to that problem. A second approach is the Virtual Log Disk [32].

Log-structured file systems (LFS) solve both the synchronous meta-data update problem and the small-write problem. Data in an LFS are coalesced and written sequentially to a segmented log. In this way, LFS avoids the seeks that a conventional file system pays in writing data back to its original location. Using this log-structured technique, LFS also solves the meta-data consistency problem by carefully ordering blocks within its segments. Like the journaling systems, LFS requires a database-like recovery phase after system crash and like Soft Updates, data are written in an order that guarantees the file system integrity. Unlike either Soft Updates or journaling, LFS requires a background garbage collector, whose performance has been the object of great speculation and debate [2][22][28][29].

Building on the idea of log-structured file systems, Wang and his colleagues propose an intelligent disk that performs writes at near maximum disk speed by selecting the destination of the write based upon the position



**Figure 5. PostMark Results.** The results are the averages of five runs; standard deviations are shown with error bars.

of the disk head [32]. The disk must then maintain a mapping of logical block numbers to physical locations. This mapping is maintained in a virtual log that is written adjacent to the actual data being written. The proposed system exists only in simulation, but seems to offer the promise of LFS-like performance for small writes, with much of the complexity hidden behind the disk interface, as is done in the AutoRaId storage system [33]. While such an approach can solve the small-write problem, it does not solve the meta-data update problem, where the file system requires that multiple related structures be consistent on disk. It does however improve the situation by allowing the synchronous writes used by FFS to occur at near maximum disk speed.

Another approach to solving the small-write problem that bears a strong resemblance to journaling is the database cache technique [6] and the more recent Disk Caching Disk (DCD) [20]. In both of these approaches, writes are written to a separate logging device, instead of being written back to the actual file system. Then, at some later point when the file system disk is not busy, the blocks are written lazily. This is essentially a two-disk journaling approach. The difference between the database cache techniques and the journaling file system technique is that the database cache tries to improve the performance of data writes as well as meta-data writes and does nothing to make meta-data operations asynchronous; instead, it makes them synchronous but with a much lower latency. In contrast, DCD places an NVRAM cache in front of the logging disk, making all small writes, including meta-data writes, asynchronous.

## 10 Conclusions

We draw several conclusions from our comparisons. At a high level, we have shown that both journaling and Soft Updates succeed at dramatically improving the performance of meta-data operations. While there are minor differences between the two journaling architectures, to a first approximation, they behave comparably. Surprisingly, we see that journaling alone is not sufficient to solve the meta-data update problem. If application and system semantics require the synchronicity of such operations, there remains a significant performance penalty, as much as 90% in some cases. In most cases, even with two disks, the penalty is substantial, unless the test was CPU-bound (e.g., the config and build phases of the `ssh` benchmark).

Soft Updates exhibits some side-effects that improve performance, in some cases significantly. Its ability to delay deletes is evidenced most clearly in the microbenchmark results. For the massive data set of the Netnews benchmark, we see that Soft Updates' ordering

constraints prevent it from achieving performance comparable to the asynchronous journaling systems, while for the small Postmark dataset, Soft Updates backgrounding of deletes provides superior performance. The race between increasing memory sizes and increasing data sets will determine which of these effects is most significant.

If our workloads are indicative of a wide range of workloads (as we hope they are), we see that meta-data operations are significant, even in CPU-dominated tasks such as the `ssh` benchmark where FFS suffers a 25% performance degradation from FFS-async. In our other test cases, the impact is even more significant (e.g., 50% for Netnews and PostMark).

The implications of such results are important as the commercial sector contemplates technology transfer from the research arena. Journaling file systems have been in widespread use in the commercial sector for many years (Veritas, IBM's JFS, Compaq's AdvFS, HP's HPFS10, Irix's XFS), while Soft Updates systems are only beginning to make an appearance. If vendors are to make informed decisions concerning the future of their file systems, analyses such as those presented here are crucial to provide the data from which to make such decisions.

## 11 Acknowledgments

We would like to thank our paper shepherd, Aaron Brown, and the anonymous reviewers for their valuable comments and suggestions. We also thank Timothy Ganger and Teagan Seltzer for their critical commentaries on the final versions of the paper.

The CMU researchers thank the members and companies of the Parallel Data Consortium (including CLARiiON, EMC, HP, Hitachi, Infineon, Intel, LSI Logic, MTI, Novell, PANASAS, Procom, Quantum, Seagate, Sun, Veritas, and 3Com) for their interest, insights, and support. They also thank IBM Corporation and CMU's Data Storage Systems Center for supporting their research efforts. The Harvard researchers thank UUNET Technologies for their support of ongoing file system research at Harvard.

## 12 References

- [1] Baker, M., Asami, S., Deprit, E., Ousterhout, J., Seltzer, M. "Non-Volatile Memory for Fast, Reliable File Systems," *Proceedings of the 5th ASPLOS*, pp. 10–22. Boston, MA, Oct. 1992.
- [2] Blackwell, T., Harris, J., Seltzer, M. "Heuristic Cleaning Algorithms in Log-Structured File Systems," *Proceedings of the 1995 USENIX Technical Conference*, pp. 277–288, New Orleans, LA, Jan. 1995.
- [3] Chen, P., Ng, W., Chandra, S., Aycock, C., Rajamani, G., Lowell, D. "The Rio File Cache: Surviving Operating



- System Crashes," *Proceedings of the 7th ASPLOS*, pp. 74–83. Cambridge, MA, Oct. 1996.
- [4] Christenson, N., Beckemeyer, D., Baker, T. "A Scalable News Architecture on a Single Spool," *login.*, 22(5), pp. 41–45. Jun. 1997.
  - [5] Chutani, S., Anderson, O., Kazer, M., Leverett, B., Mason, W.A., Sidebotham, R. "The Episode File System," *Proceedings of the 1992 Winter USENIX Technical Conference*, pp. 43–60. San Francisco, CA, Jan. 1992.
  - [6] Elkhartdt, K., Bayer, R. "A Database Cache for High Performance and Fast Restart in Database Systems," *ACM Transactions on Database Systems*, 9(4), pp. 503–525. Dec. 1984.
  - [7] Fritchie, S. "The Cyclic News Filesystem: Getting INN To Do More With Less," *Proceedings of the 1997 LISA Conference*, pp. 99–111. San Diego, CA, Oct. 1997.
  - [8] Gaede, S. "Tools for Research in Computer Workload Characterization," *Experimental Computer Performance and Evaluation*, 1981, ed Ferrari and Spadoni.
  - [9] Gaede, S. "A Scaling Technique for Comparing Interactive System Capacities," *Proceedings of the 13th International Conference on Management and Performance Evaluation of Computer Systems*, pp 62–67. 1982.
  - [10] Gaede, S. "Perspectives on the SPEC SDET Benchmark," <http://www.spec.org/osg/sdm91/sdet/index.html>.
  - [11] Ganger, G., Patt, Y. "Metadata Update Performance in File Systems," *Proceedings of the First OSDI*, pp. 49–60. Monterey, CA, Nov. 1994.
  - [12] Ganger, G., Patt Y. "Soft Updates: A Solution to the Metadata Update Problem in File Systems," Report CSE-TR-254-95. University of Michigan, Ann Arbor, MI, Aug. 1995.
  - [13] Ganger, G., Kaashoek, M.F. "Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files," *Proceedings of the 1997 USENIX Technical Conference*, pp. 1–17. Anaheim, CA, Jan. 1997.
  - [14] Ganger, G., McKusick, M.K., Soules, C., Patt, Y., "Soft Updates: A Solution to the Metadata Update Problem in File Systems," to appear in *ACM Transactions on Computer Systems*.
  - [15] Gray, J., Reuter, A. *Transaction Processing: Concepts and Techniques*. San Mateo, CA: Morgan Kaufmann, 1993.
  - [16] Hagmann, R. "Reimplementing the Cedar File System Using Logging and Group Commit," *Proceedings of the 11th SOSP*, pp. 155–162. Austin, TX, Nov. 1987.
  - [17] Haskin, R., Malachi, Y., Sawdon, W., Chan, G. "Recovery Management in QuickSilver," *ACM Transactions on Computer Systems*, 6(1), pp. 82–108. Feb. 1988.
  - [18] Hitz, D., Lau, J., Malcolm, M., "File System Design for an NFS File Server Appliance," *Proceedings of the 1994 Winter USENIX Conference*, pp. 235–246. San Francisco, CA, Jan. 1994.
  - [19] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., West, M. "Scale and Performance in a Distributed File System." *ACM Transactions on Computer Systems*, 6(1), pp. 51–81. Feb. 1988.
  - [20] Hu, Y., Yang, Q. "DCD—disk caching disk: A new approach for boosting I/O performance," *Proceedings of the 23rd ISCA*, pp. 169–178. Philadelphia, PA, May 1996.
  - [21] Katcher, J., "PostMark: A New File System Benchmark," Technical Report TR3022. Network Appliance Inc., Oct. 1997.
  - [22] Matthews, J., Roselli, D., Costello, A., Wang, R., Anderson, T. "Improving the Performance of Log-Structured File Systems with Adaptive Methods," *Proceedings of the 16th SOSP*, pp. 238–251. Saint-Malo, France, Oct. 1997.
  - [23] McKusick, M.K., Joy, W., Leffler, S., Fabry, R. "A Fast File System for UNIX," *ACM Transactions on Computer Systems* 2(3), pp 181–197. Aug. 1984.
  - [24] McKusick, M.K., Ganger, G., "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem," *Proceedings of the 1999 Freenix track of the USENIX Technical Conference*, pp. 1–17. Jun. 1999.
  - [25] McVoy, L., Kleiman, S. "Extent-like Performance From a UNIX File System," *Proceedings of the 1991 Winter USENIX Technical Conference*, pp. 33–44. Dallas, TX, Jan. 1991.
  - [26] Peacock, J.K. "The Counterpoint fast file system," *Proceedings of the 1988 Winter USENIX Technical Conference*, pp. 243–249. Dallas, TX, Feb. 1988.
  - [27] Rosenblum, M., Ousterhout, J. "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, 10(1), pp. 26–52. Feb. 1992.
  - [28] Seltzer, M., Bostic, K., McKusick, M.K., Staelin, C. "An Implementation of a Log-Structured File System for UNIX," *Proceedings of the 1993 USENIX Winter Technical Conference*, pp. 307–326. San Diego, CA, Jan. 1993.
  - [29] Seltzer, M., Smith, K., Balakrishnan, H., Chang, J., McMains, S., Padmanabhan, V. "File System Logging versus Clustering: A Performance Comparison," *Proceedings of the 1995 USENIX Technical Conference*, pp. 249–264. New Orleans, LA, Jan. 1995.
  - [30] Stein, C. "The Write-Ahead File System: Integrating Kernel and Application Logging," Harvard University Technical Report, TR-02-00, Cambridge, MA, Apr. 2000.
  - [31] Swartz, K. "The Brave Little Toaster Meets Usenet," *LISA '96*, pp. 161–170. Chicago, IL, Oct. 1996.
  - [32] Wang, R., Anderson, T., Patterson, D. "Virtual Log Based File Systems for a Programmable Disk," *Proceedings of the 3rd OSDI*, pp. 29–44. New Orleans, LA, Feb. 1999.
  - [33] Wilkes, J., Golding, R., Staelin, C., Sullivan, T. "The HP AutoRAID hierarchical storage system," *15th SOSP*, pp. 96–108. Copper Mountain, CO, Dec. 1995.
  - [34] Ylonen, T. "SSH—Secure Login Connections Over the Internet," *6th USENIX Security Symposium*, pp. 37–42. San Jose, CA, Jul. 1996.



# Lexical File Names in Plan 9 or Getting Dot-Dot Right

Rob Pike

*Bell Laboratories*  
*Murray Hill, New Jersey 07974*  
*rob@plan9.bell-labs.com*

**Abstract:** Symbolic links make the Unix file system non-hierarchical, resulting in multiple valid path names for a given file. This ambiguity is a source of confusion, especially since some shells work overtime to present a consistent view from programs such as `pwd`, while other programs and the kernel itself do nothing about the problem.

Plan 9 has no symbolic links but it does have other mechanisms that produce the same difficulty. Moreover, Plan 9 is founded on the ability to control a program's environment by manipulating its name space. Ambiguous names muddle the result of operations such as copying a name space across the network.

To address these problems, the Plan 9 kernel has been modified to maintain an accurate path name for every active file (open file, working directory, mount table entry) in the system. The definition of 'accurate' is that the path name for a file is guaranteed to be the rooted, absolute name the program used to acquire it. These names are maintained by an efficient method that combines lexical processing—such as evaluating `..` by just removing the last path name element of a directory—with local operations within the file system to maintain a consistently, easily understood view of the name system. Ambiguous situations are resolved by examining the lexically maintained names themselves.

A new kernel call, `fd2path`, returns the file name associated with an open file, permitting the use of reliable names to improve system services ranging from `pwd` to debugging. Although this work was done in Plan 9, Unix systems could also benefit from the addition of a method to recover the accurate name of an open file or the current directory.

## 1. Motivation

Consider the following unedited transcript of a session running the Bourne shell on a modern Unix system:

```
% echo $HOME
/home/rob
% cd $HOME
% pwd
/n/bopp/v7/rob
% cd /home/rob
% cd /home/ken
% cd ../rob
../rob: bad directory
%
```

(The same output results from running `tcsh`; we'll discuss `ksh` in a moment.) To a neophyte being schooled in the delights of a hierarchical file name space, this behavior must be baffling. It is, of course, the consequence of a series of symbolic links intended to give users the illusion they share a disk, when in fact their files are scattered over several devices:

```
% ls -ld /home/rob /home/ken
lrwxr-xr-x 1 root sys 14 Dec 26 1998
/home/ken -> /n/bopp/v6/ken
lrwxr-xr-x 1 root sys 14 Dec 23 1998
/home/rob -> /n/bopp/v7/rob
%
```

The introduction of symbolic links has changed the Unix file system from a true hierarchy into a directed graph, rendering `..` ambiguous and sowing confusion.

Unix popularized hierarchical naming, but the introduction of symbolic links made its naming irregular. Worse, the `pwd` command, through the underlying `getwd` library routine, uses a tricky, expensive algorithm that often delivers the wrong answer. Starting from the current directory, `getwd` opens the parent, `..`, and searches it for an entry whose `i`-number matches the current directory; the matching entry is the final path element of the ultimate result. Applying this process iteratively, `getwd` works back towards the root. Since `getwd` knows nothing about symbolic links, it will recover surprising names for directories reached by them, as illustrated by the example; the backward paths `getwd` traverses will not backtrack across the links.

Partly for efficiency and partly to make `cd` and `pwd` more predictable, the Korn shell `ksh` [Korn94] implements `pwd` as a builtin. (The `cd` command must be a builtin in any shell, since the current directory is unique to each process.) `Ksh` maintains its own private view of the file system to try to disguise symbolic links; in particular, `cd` and `pwd` involve some lexical processing (somewhat like the `cleanname` function discussed later in this paper), augmented by heuristics such as examining the environment for names like `$HOME` and `$PWD` to assist initialization of the state of the private view. [Korn00]

This transcript begins with a Bourne shell running:

```
% cd /home/rob
% pwd
/n/bopp/v7/rob
% ksh
$ pwd
/home/rob
$
```

This result is encouraging. Another example, again starting from a Bourne shell:

```
% cd /home/rob
% cd ../ken
../ken: bad directory
% ksh
$ pwd
/home/rob
$ cd ../ken
$ pwd
/home/ken
$
```

By doing extra work, the Korn shell is providing more sensible behavior, but it is easy to defeat:

```
% cd /home/rob
% pwd
/n/bopp/v7/rob
% cd bin
% pwd
/n/bopp/v7/rob/bin
% ksh
$ pwd
/n/bopp/v7/rob/bin
$ exit
% cd /home/ken
% pwd
/n/bopp/v6/ken
% ksh
$ pwd
/n/bopp/v6/ken
$
```

In these examples, `ksh`'s built-in `pwd` failed to produce the results (`/home/rob/bin` and `/home/ken`) that the previous example might have led us to expect. The Korn shell is hiding the problem, not solving it, and in fact is not even hiding it very well.

A deeper question is whether the shell should even be trying to make `pwd` and `cd` do a better job. If it does, then the `getwd` library call and every program that uses it will behave differently from the shell, a situation that is sure to confuse. Moreover, the ability to change directory to `../ken` with the Korn shell's `cd` command but not with the `chdir` system call is a symptom of a diseased system, not a healthy shell.

The operating system should provide names that work and make sense. Symbolic links, though, are here to stay, so we need a way to provide sensible, unambiguous names in the face of a non-hierarchical name space. This paper shows how the challenge was met on Plan 9, an operating system with Unix-like naming.

## 2. Names in Plan 9

Except for some details involved with bootstrapping, file names in Plan 9 have the same syntax as in Unix. Plan 9 has no symbolic links, but its name space construction operators, `bind` and `mount`, make it possible to build the same sort of non-hierarchical structures created by symbolically linking directories on Unix.

Plan 9's `mount` system call takes a file descriptor and attaches to the local name space the file system service it represents:

```
mount(fd, "/dir", flags)
```

Here `fd` is a file descriptor to a communications port such as a pipe or network connection; at the other end of the port is a service, such as file server, that talks 9P, the Plan 9 file system protocol. After the call succeeds, the root directory of the service will be visible at the *mount point* `/dir`, much as with the `mount` call of Unix. The `flag` argument specifies the nature of the attachment: `MREPL` says that the contents of the root directory (appear to) replace the current contents of `/dir`; `MAFTER` says that the current contents of `dir` remain visible, with the mounted directory's contents appearing *after* any existing files; and `MBEFORE` says that the contents remain visible, with the mounted directory's contents appearing *before* any existing files. These multicomponent directories are called *union directories* and are somewhat different from union directories in 4.4BSD-Lite [PeMc95], because only the top-level directory itself is unioned, not its descendents, recursively. (Plan 9's union directories are used differently from 4.4BSD-Lite's, as will become apparent.)

For example, to bootstrap a diskless computer the system builds a local name space containing only the root directory, `/`, then uses the network to open a connection to the main file server. It then executes

```
mount(rootfd, "/", MREPL);
```

After this call, the entire file server's tree is visible, starting from the root of the local machine.

While `mount` connects a new service to the local name space, `bind` rearranges the existing name space:

```
bind("tofile", "fromfile", flags)
```

causes subsequent mention of the `fromfile` (which may be a plain file or a directory) to behave as though `tofile` had been mentioned instead, somewhat like a symbolic link. (Note, however, that the arguments are in the opposite order compared to `ln -s`). The `flags` argument is the same as with `mount`.

As an example, a sequence something like the following is done at bootstrap time to assemble, under the single directory `/bin`, all of the binaries suitable for this architecture, represented by (say) the string `sparc`:

```
bind("/sparc/bin", "/bin",  
    MREPL);  
bind("/usr/rob/sparc/bin", "/bin",  
    MAFTER);
```

This sequence of binds causes `/bin` to contain first the standard binaries, then the contents of `rob`'s private SPARC binaries. The ability to build such union directories obviates the need for a shell `$PATH` variable while providing opportunities for managing heterogeneity. If the system were a Power PC, the same sequence would be run with `power` textually substituted for `sparc` to place the Power PC binaries in `/bin` rather than the SPARC binaries.

Trouble is already brewing. After these bindings are set up, where does

```
% cd /bin  
% cd ..
```

set the current working directory, to `/` or `/sparc` or `/usr/rob/sparc`? We will return to this issue.

There are some important differences between binds and symbolic links. First, symbolic links are a static part of the file system, while Plan 9 bindings are created at run time, are stored in the kernel, and endure only as long as the system maintains them; they are temporary. Since they are known to the kernel but not the file system, they must be set up each time the kernel boots or a user logs in; permanent bindings are created by editing system initialization scripts and user profiles rather than by building them in the file system itself.

The Plan 9 kernel records what bindings are active for a process, whereas symbolic links, being held on the Unix file server, may strike whenever the process evaluates a file name. Also, symbolic links apply to all processes that evaluate the affected file, whereas `bind` has

a local scope, applying only to the process that executes it and possibly some of its peers, as discussed in the next section. Symbolic links cannot construct the sort of `/bin` directory built here; it is possible to have multiple directories point to `/bin` but not the other way around.

Finally, symbolic links are symbolic, like macros: they evaluate the associated names each time they are accessed. Bindings, on the other hand, are evaluated only once, when the `bind` is executed; after the binding is set up, the kernel associates the underlying files, rather than their names. In fact, the kernel's representation of a `bind` is identical to its representation of a `mount`; in effect, a `bind` is a `mount` of the `tofile` upon the `fromfile`. The binds and mounts coexist in a single *mount table*, the subject of the next section.

### 3. The Mount Table

Unix has a single global mount table for all processes in the system, but Plan 9's mount tables are local to each process. By default it is inherited when a process forks, so mounts and binds made by one process affect the other, but a process may instead inherit a copy, so modifications it makes will be invisible to other processes. The convention is that related processes, such as processes running in a single window, share a mount table, while sets of processes in different windows have distinct mount tables. In practice, the name spaces of the two windows will appear largely the same, but the possibility for different processes to see different files (hence services) under the same name is fundamental to the system, affecting the design of key programs such as the window system [Pike91].

The Plan 9 mount table is little more than an ordered list of pairs, mapping the `fromfiles` to the `tofiles`. For mounts, the `tofile` will be an item called a `Channel`, similar to a Unix `vnode`, pointing to the root of the file service, while for a `bind` it will be the `Channel` pointing to the `tofile` mentioned in the `bind` call. In both cases, the `fromfile` entry in the table will be a `Channel` pointing to the `fromfile` itself.

The evaluation of a file name proceeds as follows. If the name begins with a slash, start with the `Channel` for the root; otherwise start with the `Channel` for the current directory of the process. For each path element in the name, such as `usr` in `/usr/rob`, try to 'walk' the `Channel` to that element [Pike93]. If the walk succeeds, look to see if the resulting `Channel` is the same as any `fromfile` in the mount table, and if so, replace it by the corresponding `tofile`. Advance to the next element and continue.



There are a couple of nuances. If the directory being walked is a union directory, the walk is attempted in the elements of the union, in order, until a walk succeeds. If none succeed, the operation fails. Also, when the destination of a walk is a directory for a purpose such as the `chdir` system call or the `fromfile` in a `bind`, once the final walk of the sequence has completed the operation stops; the final check through the mount table is not done. Among other things, this simplifies the management of union directories; for example, subsequent `bind` calls will append to the union associated with the underlying `fromfile` instead of what is bound upon it.

#### 4. A Definition of Dot-Dot

The ability to construct union directories and other intricate naming structures introduces some thorny problems: as with symbolic links, the name space is no longer hierarchical, files and directories can have multiple names, and the meaning of `..`, the parent directory, can be ambiguous.

The meaning of `..` is straightforward if the directory is in a locally hierarchical part of the name space, but if we ask what `..` should identify when the current directory is a mount point or union directory or multiply symlinked spot (which we will henceforth call just a mount point, for brevity), there is no obvious answer. Name spaces have been part of Plan 9 from the beginning, but the definition of `..` has changed several times as we grappled with this issue. In fact, several attempts to clarify the meaning of `..` by clever coding resulted in definitions that could charitably be summarized as ‘what the implementation gives.’

Frustrated by this situation, and eager to have better-defined names for some of the applications described later in this paper, we recently proposed the following definition for `..`:

The parent of a directory `X`, `X/..`, is the same directory that would obtain if we instead accessed the directory named by stripping away the last path name element of `X`.

For example, if we are in the directory `/a/b/c` and `chdir` to `..`, the result is *exactly* as if we had executed a `chdir` to `/a/b`.

This definition is easy to understand and seems natural. It is, however, a purely *lexical* definition that flatly ignores evaluated file names, mount tables, and other kernel-resident data structures. Our challenge is to implement it efficiently. One obvious (and correct) implementation is to rewrite path names lexically to fold out `..`, and then evaluate the file name forward from the root, but this is expensive and unappealing.

We want to be able to use local operations to evaluate file names, but maintain the global, lexical definition of dot-dot. It isn't too hard.

#### 5. The Implementation

To operate lexically on file names, we associate a name with each open file in the kernel, that is, with each `Channel` data structure. The first step is therefore to store a `char*` with each `Channel` in the system, called its `Cname`, that records the *absolute* rooted file name for the `Channel`. `Cnames` are stored as full text strings, shared copy-on-write for efficiency. The task is to maintain each `Cname` as an accurate absolute name using only local operations.

When a file is opened, the file name argument in the open (or `chdir` or `bind` or ...) call is recorded in the `Cname` of the resulting `Channel`. When the file name begins with a slash, the name is stored as is, subject to a cleanup pass described in the next section. Otherwise, it is a local name, and the file name must be made absolute by prefixing it with the `Cname` of the current directory, followed by a slash. For example, if we are in `/home/rob` and `chdir` to `bin`, the `Cname` of the resulting `Channel` will be the string `/home/rob/bin`.

This assumes, of course, that the local file name contains no `..` elements. If it does, instead of storing for example `/home/rob/..` we delete the last element of the existing name and set the `Cname` to `/home`. To maintain the lexical naming property we must guarantee that the resulting `Cname`, if it were to be evaluated, would yield the identical directory to the one we actually do get by the local `..` operation.

If the current directory is not a mount point, it is easy to maintain the lexical property. If it is a mount point, though, it is still possible to maintain it on Plan 9 because the mount table, a kernel-resident data structure, contains all the information about the non-hierarchical connectivity of the name space. (On Unix, by contrast, symbolic links are stored on the file server rather than in the kernel.) Moreover, the presence of a full file name for each `Channel` in the mount table provides the information necessary to resolve ambiguities.

The mount table is examined in the `from`→`to` direction when evaluating a name, but `..` points backwards in the hierarchy, so to evaluate `..` the table must be examined in the `to`→`from` direction. (“How did we get here?”)

The value of `..` is ambiguous when there are multiple bindings (mount points) that point to the directories involved in the evaluation of `..`. For example, return



to our original script with `/n/bopp/v6` (containing a home directory for ken) and `/n/bopp/v7` (containing a home directory for rob) unioned into `/home`. This is represented by two entries in the mount table, `from=/home, to=/n/bopp/v6` and `from=/home, to=/n/bopp/v7`. If we have set our current directory to `/home/rob` (which has landed us in the physical location `/n/bopp/v7/rob`) our current directory is not a mount point but its parent is. The value of `..` is ambiguous: it could be `/home`, `/n/bopp/v7`, or maybe even `/n/bopp/v6`, and the ambiguity is caused by two `tofiles` bound to the same `fromfile`. By our definition, if we now evaluate `..`, we should acquire the directory `/home`; otherwise `../ken` could not possibly result in ken's home directory, which it should. On the other hand, if we had originally gone to `/n/bopp/v7/rob`, the name `../ken` should *not* evaluate to ken's home directory because there is no directory `/n/bopp/v7/ken` (ken's home directory is on `v6`). The problem is that by using local file operations, it is impossible to distinguish these cases: regardless of whether we got here using the name `/home/rob` or `/n/bopp/v7/rob`, the resulting directory is the same. Moreover, the mount table does not itself have enough information to disambiguate: when we do a local operation to evaluate `..` and land in `/n/bopp/v7`, we discover that the directory is a `tofile` in the mount table; should we step back through the table to `/home` or not?

The solution comes from the `Cnames` themselves. Whether to step back through the mount point `from=/home, to=/n/bopp/v7` when evaluating `..` in rob's directory is trivially resolved by asking the question, Does the `Cname` for the directory begin `/home`? If it does, then the path that was evaluated to get us to the current directory must have gone through this mount point, and we should back up through it to evaluate `..`; if not, then this mount table entry is irrelevant.

More precisely, both *before* and *after* each `..` element in the path name is evaluated, if the directory is a `tofile` in the mount table, the corresponding `fromfile` is taken instead, provided the `Cname` of the corresponding `fromfile` is the prefix of the `Cname` of the original directory. Since we always know the full name of the directory we are evaluating, we can always compare it against all the entries in the mount table that point to it, thereby resolving ambiguous situations and maintaining the lexical property of `..`. This check also guarantees we don't follow a misleading mount point, such as the entry pointing to `/home` when we are really in `/n/bopp/v7/rob`. Keeping the full names with the Channels makes it

easy to use the mount table to decide how we got here and, therefore, how to get back.

In summary, the algorithm is as follows. Use the usual file system operations to walk to `..`; call the resulting directory `d`. Lexically remove the last element of the initial file name. Examine all entries in the mount table whose `tofile` is `d` and whose `fromfile` has a `Cname` identical to the truncated name. If one exists, that `fromfile` is the correct result; by construction, it also has the right `Cname`. In our example, evaluating `..` in `/home/rob` (really `/n/bopp/v7/rob`) will set `d` to `/n/bopp/v7`; that is a `tofile` whose `fromfile` is `/home`. Removing the `/rob` from the original `Cname`, we find the name `/home`, which matches that of the `fromfile`, so the result is the `fromfile`, `/home`.

Since this implementation uses only local operations to maintain its names, it is possible to confuse it by external changes to the file system. Deleting or renaming directories and files that are part of a `Cname`, or modifying the mount table, can introduce errors. With more implementation work, such mistakes could probably be caught, but in a networked environment, with machines sharing a remote file server, renamings and deletions made by one machine may go unnoticed by others. These problems, however, are minor, uncommon and, most important, easy to understand. The method maintains the lexical property of file names unless an external agent changes the name surreptitiously; within a stable file system, it is always maintained and `pwd` is always right.

To recapitulate, maintaining the Channel's absolute file names lexically and using the names to disambiguate the mount table entries when evaluating `..` at a mount point combine to maintain the lexical definition of `..` efficiently.

## 6. Cleaning names

The lexical processing can generate names that are messy or redundant, ones with extra slashes or embedded `../` or `./` elements and other extraneous artifacts. As part of the kernel's implementation, we wrote a procedure, `cleannname`, that rewrites a name in place to canonicalize its appearance. The procedure is useful enough that it is now part of the Plan 9 C library and is employed by many programs to make sure they always present clean file names.

`Cleannname` is analogous to the URL-cleaning rules defined in RFC 1808 [Field95], although the rules are slightly different. `Cleannname` iteratively does the following until no further processing can be done:

1. Reduce multiple slashes to a single slash.

2. Eliminate . path name elements (the current directory).
3. Eliminate .. path name elements (the parent directory) and the non-. non-.., element that precedes them.
4. Eliminate .. elements that begin a rooted path, that is, replace /.. by / at the beginning of a path.
5. Leave intact .. elements that begin a non-rooted path.

If the result of this process is a null string, `cleannname` returns the string ".", representing the current directory.

### 7. The `fd2path` system call

Plan 9 has a new system call, `fd2path`, to enable programs to extract the Cname associated with an open file descriptor. It takes three arguments: a file descriptor, a buffer, and the size of the buffer:

```
int fd2path(int fd, char *buf, int nbuf)
```

It returns an error if the file descriptor is invalid; otherwise it fills the buffer with the name associated with `fd`. (If the name is too long, it is truncated; perhaps this condition should also draw an error.) The `fd2path` system call is very cheap, since all it does is copy the Cname string to user space.

The Plan 9 implementation of `getwd` uses `fd2path` rather than the tricky algorithm necessary in Unix:

```
char*
getwd(char *buf, int nbuf)
{
    int n, fd;

    fd = open(".", OREAD);
    if(fd < 0)
        return NULL;
    n = fd2path(fd, buf, nbuf);
    close(fd);
    if(n < 0)
        return NULL;
    return buf;
}
```

(The Unix specification of `getwd` does not include a count argument.) This version of `getwd` is not only straightforward, it is very efficient, reducing the performance advantage of a built-in `pwd` command while guaranteeing that all commands, not just `pwd`, see sensible directory names.

Here is a routine that prints the file name associated with each of its open file descriptors; it is useful for

tracking down file descriptors left open by network listeners, text editors that spawn commands, and the like:

```
#define NBUF 256

void
openfiles(void)
{
    int i;
    char buf[NBUF];

    for(i=0; i<NFD; i++)
        if(fd2path(i, buf, NBUF) >= 0)
            print("%d: %s\n", i, buf);
}
```

### 8. Uses of good names

Although `pwd` was the motivation for getting names right, good file names are useful in many contexts and have become a key part of the Plan 9 programming environment. The compilers record in the symbol table the full name of the source file, which makes it easy to track down the source of buggy, old software and also permits the implementation of a program, `src`, to automate tracking it down. Given the name of a program, `src` reads its symbol table, extracts the file information, and triggers the editor to open a window on the program's source for its main routine. No guesswork, no heuristics.

The `openfiles` routine was the inspiration for a new file in the `/proc` file system [Kill84]. For process `n`, the file `/proc/n/fd` is a list of all its open files, including its working directory, with associated information including its open status, I/O offset, unique id (analogous to i-number) and file name. Figure 1 shows the contents of the `fd` file for a process in the window system on the machine being used to write this paper.

(The Linux implementation of `/proc` provides a related service by giving a directory in which each file-descriptor-numbered file is a symbolic link to the file itself.) When debugging errant systems software, such information can be valuable.

Another motivation for getting names right was the need to extract from the system an accurate description of the mount table, so that a process's name space could be recreated on another machine, in order to move (or simulate) a computing environment across the network. One program that does this is Plan 9's `cpu` command, which recreates the local name space on a remote machine, typically a large fast multiprocessor. Without accurate names, it was impossible to do the job right; now `/proc` provides a description of the name space of each process, `/proc/n/ns`:

```
% cat /proc/125099/fd
/usr/rob
0 r M 5141 00000001.00000000 0 /mnt/term/dev/cons
1 w M 5141 00000001.00000000 51 /mnt/term/dev/cons
2 w M 5141 00000001.00000000 51 /mnt/term/dev/cons
3 r M 5141 0000000b.00000000 1166 /dev/snarf
4 rw M 5141 0ffffffc.00000000 288 /dev/draw/new
5 rw M 5141 00000036.00000000 4266337 /dev/draw/3/data
6 r M 5141 00000037.00000000 0 /dev/draw/3/refresh
7 r c 0 00000004.00000000 6199848 /dev/bintime
%
```

Figure 1. The contents of the fd (open file descriptor) file.

```
% cat /proc/125099/ns
bind / /
mount -aC #s/boot /
bind #c /dev
bind #d /fd
bind -c #e /env
bind #p /proc
bind -c #s /srv
bind /386/bin /bin
bind -a /rc/bin /bin
bind /net /net
bind -a #l /net
mount -a #s/cs /net
mount -a #s/dns /net
bind -a #D /net
mount -c #s/boot /n/emelie
bind -c /n/emelie/mail /mail
mount -c /net/il/134/data /mnt/term
bind -a /usr/rob/bin/rc /bin
bind -a /usr/rob/bin/386 /bin
mount #s/boot /n/emelieother other
bind -c /n/emelieother/rob /tmp
mount #s/boot /n/dump dump
bind /mnt/term/dev/cons /dev/cons
...
cd /usr/rob
%
```

(The # notation identifies raw device drivers so they may be attached to the name space.) The last line of the file gives the working directory of the process. The format of this file is that used by a library routine, `newns`, which reads a textual description like this and reconstructs a name space. Except for the need to quote # characters, the output is also a shell script that invokes the user-level commands `bind` and `mount`, which are just interfaces to the underlying system calls. However, files like `/net/il/134/data` represent network connections; to find out where they point, so that the corresponding calls can be reestablished for another process, they must be examined in more detail using the network device files [PrWi93]. Another program, `ns`, does this; it reads the `/proc/n/ns` file, decodes the information, and interprets it, translating the network addresses and quoting the names when required:

```
...
mount -a '#s/dns' /net
...
mount -c il!135.104.3.100!12884 /mnt/term
...
```

These tools make it possible to capture an accurate description of a process's name space and recreate it elsewhere. And like the open file descriptor table, they are a boon to debugging; it is always helpful to know exactly what resources a program is using.

## 9. Adapting to Unix

This work was done for the Plan 9 operating system, which has the advantage that the non-hierarchical aspects of the name space are all known to the kernel. It should be possible, though, to adapt it to a Unix system. The problem is that Unix has nothing corresponding precisely to a Channel, which in Plan 9 represents the unique result of evaluating a name. The `vnode` structure is a shared structure that may represent a file known by several names, while the `file` structure refers only to open files, but for example the current working directory of a process is not open. Possibilities to address this discrepancy include introducing a Channel-like structure that connects a name and a `vnode`, or maintaining a separate per-process table that maps names to `vnodes`, disambiguating using the techniques described here. If it could be done the result would be an implementation of ... that reduces the need for a built-in `pwd` in the shell and offers a consistent, sensible interpretation of the 'parent directory'.

We have not done this adaptation, but we recommend that the Unix community try it.

## 10. Conclusions

It should be easy to discover a well-defined, absolute path name for every open file and directory in the system, even in the face of symbolic links and other non-hierarchical elements of the file name space. In earlier versions of Plan 9, and all current versions of Unix, names can instead be inconsistent and confusing.

The Plan 9 operating system now maintains an accurate name for each file, using inexpensive lexical operations coupled with local file system actions. Ambiguities are resolved by examining the names themselves; since they reflect the path that was used to reach the file, they also reflect the path back, permitting a dependable answer to be recovered even when stepping backwards through a multiply-named directory.

Names make sense again: they are sensible and consistent. Now that dependable names are available, system services can depend on them, and recent work in Plan 9 is doing just that. We—the community of Unix and Unix-like systems—should have done this work a long time ago.

## 11. Acknowledgements

Phil Winterbottom devised the `ns` command and the `fd` and `ns` files in `/proc`, based on an earlier implementation of path name management that the work in this paper replaces. Russ Cox wrote the final version of `cleannname` and helped debug the code for reversing the mount table. Ken Thompson, Dave Presotto, and Jim McKie offered encouragement and consultation.

## 12. References

- [Field95] R. Fielding, “Relative Uniform Resource Locators”, *Network Working Group Request for Comments: 1808*, June, 1995.
- [Kill84] T. J. Killian, “Processes as Files”, *Proceedings of the Summer 1984 USENIX Conference*, Salt Lake City, 1984, pp. 203-207.
- [Korn94] David G. Korn, “ksh: An Extensible High Level Language”, *Proceedings of the USENIX Very High Level Languages Symposium*, Santa Fe, 1994, pp. 129-146.
- [Korn00] David G. Korn, personal communication.
- [PeMc95] Jan-Simon Pendry and Marshall Kirk McKusick, “Union Mounts in 4.4BSD-Lite”, *Proceedings of the 1995 USENIX Conference*, New Orleans, 1995.
- [Pike91] Rob Pike, “8½, the Plan 9 Window System”, *Proceedings of the Summer 1991 USENIX Conference*, Nashville, 1991, pp. 257-265.
- [Pike93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom, “The Use of Name Spaces in Plan 9”, *Operating Systems Review*, 27, 2, April 1993, pp. 72-76.
- [PrWi93] Dave Presotto and Phil Winterbottom, “The Organization of Networks in Plan 9”, *Proceedings of the Winter 1993 USENIX Conference*, San Diego, 1993, pp. 43-50.



# Gecko: tracking a very large billing system

*Andrew Hume*

AT&T Labs - Research  
andrew@research.att.com

*Scott Daniels*

Electronic Data Systems Corporation  
scott.daniels@ti.com

*Angus MacLellan*

AT&T Labs - Research  
amaclellan@ems.att.com

## Abstract

There is a growing need for very large databases which are not practical to implement with conventional relational database technology. These databases are characterised by huge size and frequent large updates; they do not require traditional database transactions, instead the atomicity of bulk updates can be guaranteed outside of the database. Given the I/O and CPU resources available on modern computer systems, it is possible to build these huge databases using simple flat files and simply scanning all the data when doing queries. This paper describes Gecko, a system for tracking the state of every call in a very large billing system, which uses sorted flat files to implement a database of about 60G records occupying 2.6TB.

This paper describes Gecko's architecture, both data and process, and how we handle interfacing with the existing legacy MVS systems. We focus on the performance issues, particularly with regard to job management, I/O management and data distribution, and on the tools we built. We finish with the important lessons we learned along the way, some tools we developed that would be useful in dealing with legacy systems, a benchmark comparing some alternative system architectures, and an assessment of the scalability of the system.

## 1. Introduction

Like most large companies, AT&T is under growing pressure to take advantage the data it collects while conducting its business. Attempts to do this with call detail (and in the near future, IP usage) are hampered by the technical challenge of dealing with very large volumes of data. Conventional databases are not able to handle such volumes, particularly when updates are frequent, mostly because of the overhead of performing these updates as transactions [Kor86]. (In the following, many of the names that follow, such as RAMP, are acronyms. Most of the time, the expanded version of the acronym is both obscure and

unilluminating; we therefore will treat them simply as names. On the other hand, Gecko is not an acronym; it's simply a type of lizard.)

Three existing examples show a range of solutions. The bill history database, used by customer care to access the last few months of bills for residential customers billed by RAMP, uses conventional database technology and massive parallelism (many thousands of instances of IMS databases) and handles about 25% of AT&T's daily call detail volume. The SCAMP project, part of a fraud detection system, uses the Daytona database[Gre99] to maintain 63 days of full volume call detail (250-300M calls/day).

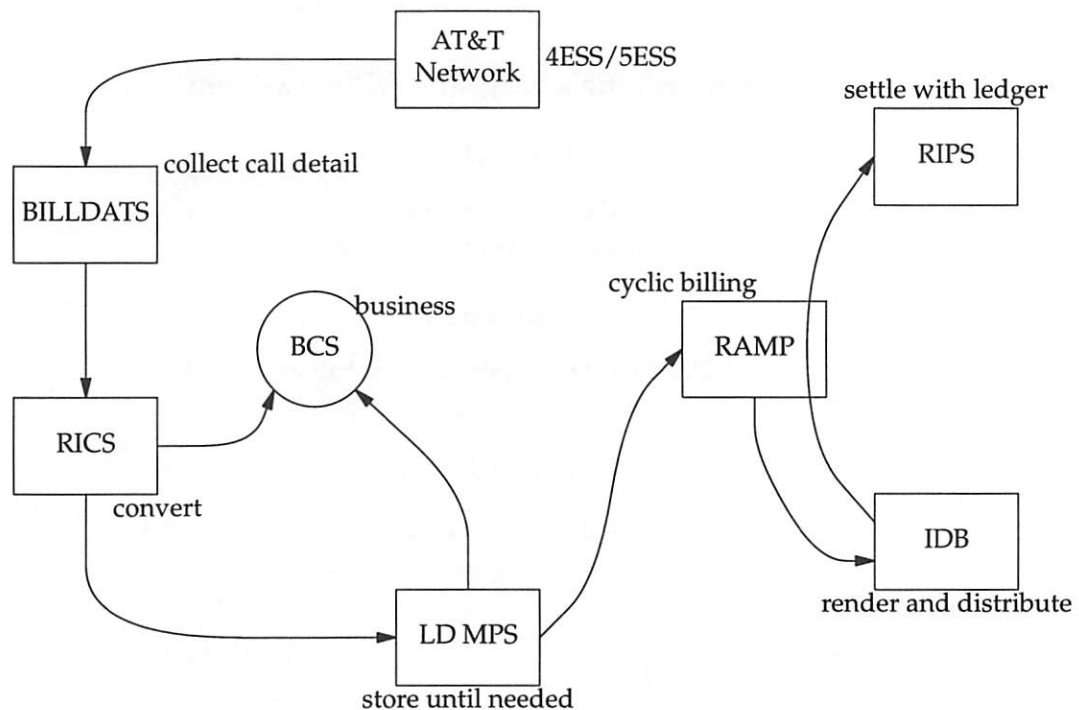


Figure 1: High level billing data flow

Each call is represented by a modest (28 fields) fixed sized record. Finally, Greyhound uses a flat file scheme to store call detail and customer account information which is then used to build various aggregate data, such as marketing and segmentation models, for marketing.

This problem, to track every message as it went through the billing process, all the way from recording through to settlement, was first raised with us in late 1995. This is an extremely hard problem; not only are the volumes huge (about four times the daily call volume), but there is no simple way to correlate the various records (from different systems) for a message. After a team at Research, including two interns from Consumer Billing, built a successful prototype in 1996, the decision was made to build a production version. A team of six people (within Consumer Billing) started in March 1997 and the system went live in December 1997.

## 2. The problem

### The business problem

The flow of records through AT&T's billing systems, from recording to settlement, is fairly complicated. Even for simple residential billed calls, the records flow through seven major systems (see figure 1) and are processed by a few hundred different processing steps. There is considerable churn both at the process level, and at the architectural level. In the face of this complexity and change, how do we know that every call is billed exactly once? This is the business question that Gecko answers.

Gecko attacks this question in a novel way: it tracks the progress of (or records corresponding to) each call throughout the billing process by tapping the dataflows between systems and within systems. Although this seems an obvious solution, the volumes involved have hitherto made this sort of scheme infeasible.

### The technical problem

The problem is threefold: we need to convert the various dataflow taps into a canonical fixed-length form (*tags*), we need to match tags for a call together into *tagsets* and maintain them in a datastore (or database),

and we need to generate various reports from the datastore. A tagset consists of one or more tags sharing the same key (24 bytes including fields like originating number and timestamp). Each day, we add tags generated from new tap files, age off certain tagsets (generally these are tagsets that have been completely processed), and generate various reports about the tagsets in the datastore. The relevant quantitative information is

- there are now about 3100 tap files per day, totaling around 240GB, producing about 1.2B tags.
- each tag is 96 bytes; a tagset is  $80+24n$  bytes (where the tagset has  $n$  tags).
- the datastore will typically contain about 60B tags in 13B tagsets.
- the target for producing reports is about 11 hours; the target for the entire cycle is about 15 hours (allowing some time for user ad hoc queries), with a maximum of 20 hours (allowing some time for system maintenance).

Tagsets which have exited the billing process, for example, calls that have been billed, are eventually aged out of the datastore. Typically, we keep tagsets for 30 days after they exit (in order to facilitate analysis).

Finally, there needs to be a mechanism for examining the tagsets contributing to any particular numeric entry in the reports; for example, if we report that 20,387 tagsets are delayed and are currently believed to be inside RICS, the users need to be able examine those tagsets.

### 3. The current architecture

The overall architecture had two main drivers: minimising the impact on the existing internal computing and networking infrastructure, and the inability of conventional database technology to handle our problem.

Our internal network support was apprehensive about Gecko because of its prodigious data transmission requirements. It was felt that adding an extra 200GB per day to the existing load, most of it long haul, was not feasible; it is close to 15% of the total network traffic. The data does compress well but it takes CPU resources to do the compression, and because our internal computing charges are based largely on CPU usage,

Gecko would end up paying an awful lot. (Actually, the project could not survive the MVS cost of data compression and would have been cancelled.) There is a loophole where small systems are billed at a flat monthly rate. Thus, we have a satellite/central server design where uncompressed data is sent from the tapped systems over a LAN to a local Gecko system (satellite) which compresses the tap data and then transmits it to the central server.

The design we implemented to solve the database problem does not use conventional database technology; as described in [Hum99], we experimented with an Oracle-based implementation, but it was unsatisfactory. The best solution only stored the last state for a call, and not all the states, and even then, the daily update cycle took 16 hours. Backup at that time was horrendous, although better solutions exist now. Finally, the database scheme depended intimately on the desired reports; if the reports changed significantly, you would likely have to redo the whole database design.

Instead, we used sorted flat files and relied on the speed and I/O capacity of modern high-end Unix systems, such as large SGI and Sun systems.

The following description reflects our current implementation; in some cases, as described in section 6, this was rather different than our original design.

**3.1 High level system design:** Gecko is constructed from three systems, as shown in figure 2. Two systems, *dtella* (in Alpharetta) and *tokay* (in Kansas City), are simple buffer systems; they receive files from local legacy systems, compress them, and then transmit them to the central system *goldeye*. These "tap files" are received into the *loading dock*, which does integrity and completeness checks, makes archival copies on tape, and then creates tags which are put into the *tag cache*. Finally, each file in the tag cache is split up into a subfile in each of the filesystems making up the datastore. The processing for a file is scheduled when it arrives, which can be 24 hours/day.

Once a day, currently at 00:30, we perform an update cycle which involves taking all the tag files that have been split and adding them to the datastore.

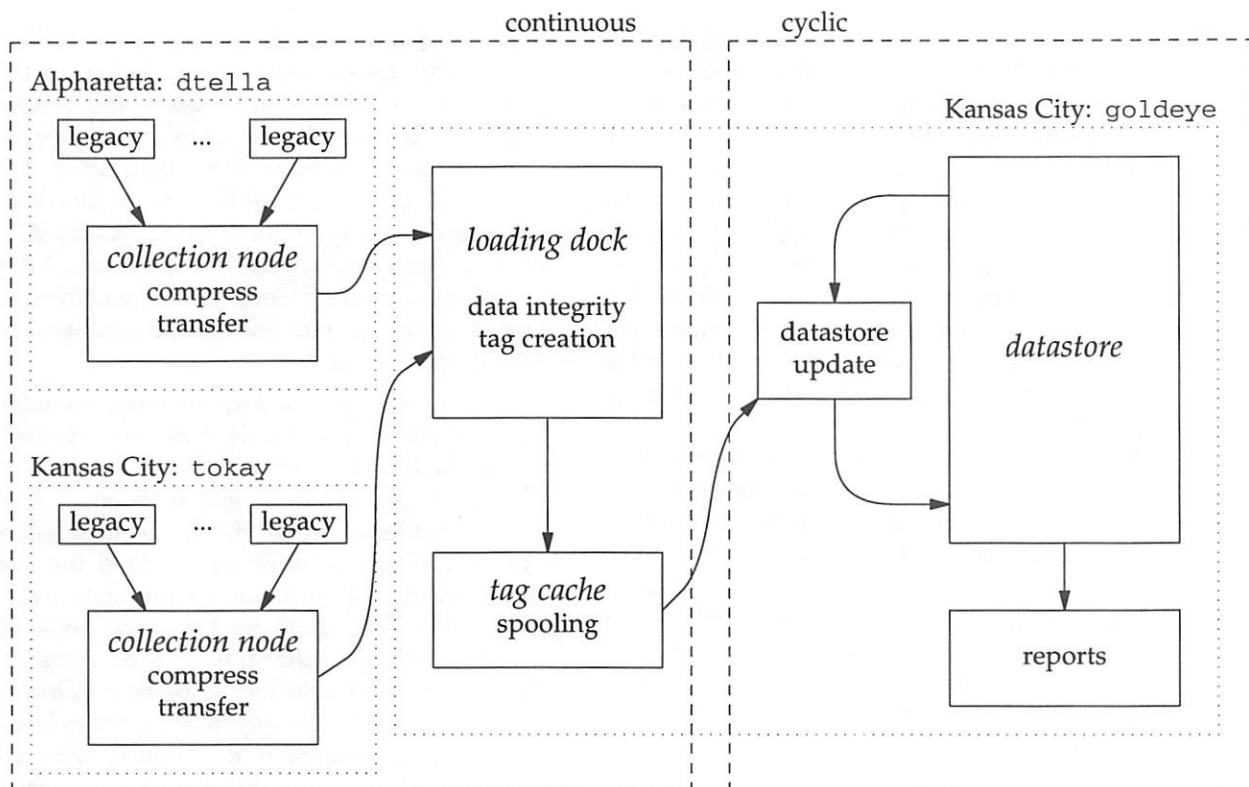


Figure 2: High level logical and system design

After all the datastore has been updated, we generate the reports.

**3.2 Data design:** The system supporting the datastore is a Sun E10000, with 32 processors and 6GB of memory, running Solaris 2.6. The datastore disk storage is provided by 16 A3000 (formerly RSM2000) RAID cabinets, which provides about 3.6TB of RAID-5 disk storage. For backup purposes, we have a StorageTek 9310 Powderhorn tape silo with 8 Redwood tape drives. Each drive can read and write at about 10MB/s. The silo has a capacity of 6000 50GB tapes.

The datastore is organised as 93 filesystems, each with 52 directories; each directory contains a *partition* of the datastore (the various magic numbers here are explained in section 3.5). Tagsets are allocated to one of these 4836 partitions according to a hash function based on the originating telephone number. Currently, the datastore is about 2.6TB. Because of Solaris's inability to sustain large amounts of sequential file I/O through the buffer cache, all the datastore filesystems are mounted as direct I/O; that is, all file I/O on those filesystems bypasses the buffer cache.

This 'feature' turned out to be a blessing in disguise because it helped us discover an unexpected and deep design paradigm: designing for a scalable cluster of systems networked together is isomorphic to designing for a single system with a scalable number of filesystems. Just as with a cluster of systems, where you try to do nearly all the work locally on each system and minimise the inter-system communications, you arrange that processing of data on each filesystem is independent of processing on any other filesystem. The goal, of course, is to make the design scalable and have predictable performance. In this case, using the system-wide buffer cache would be an unnecessary bottleneck. This isomorphism is so pervasive that when we evaluate design changes, we think of filesystems as having bandwidths and copying files from one filesystem to another is exactly the same as *ftp*ing files over a network.

This isomorphism seems related to the duality discussed in [Lau79], a duality between systems made of a smaller number of larger processes communicating by



modest numbers of messages and systems comprising large numbers of small processes communicating via shared memory. By replacing 'shared memory' by 'intrasystem file I/O' and 'message passing' with 'networked file I/O', Lauer and Needham's arguments about the fundamental equivalence of the two approaches seem fresh and persuasive.

It also helps with extracting maximal performance out of each filesystem, which depends, and has always depended, on minimising the amount of disk head movement. We did this by two design principles: accessing data sequentially rather than random access, and carefully controlling the amount of simultaneous access to the filesystem. The implementation also needs care; the two most important techniques are strict adherence to a good set of I/O block sizes (we use 256KB chosen to match the stripe width of the RAID-5 configuration of the underlying storage array), and using multiple buffers so that there is always an I/O request outstanding (we use an internally developed buffering scheme, based on POSIX threads, that currently uses 3 readers or writers).

Controlling the amount of simultaneous access to each filesystem was easy because we use a single tool, *woomera*, to control the over 10,000 jobs needed to update the datastore. The relevant part of *woomera*, which is described in more detail in section 5.4, is that jobs can be marked as using various resources and it is easy to specify limits on how many jobs sharing a resource may run simultaneously. By marking all the jobs that access filesystem *gb* with a resource *LVOLgb*, we can set the limit for the resource *LVOLgb* to one and thus ensure that at most one job will access that filesystem at any given time.

**3.3 Report architecture:** Gecko is required to generate three different reports. Two of the reports describe calls that are still being processed, and the other describes the eventual disposition of calls. This latter report requires a summary of all calls ever processed. The current reporting architecture is a combination of two things. One, the history file, is a summary of all tagsets that have been deleted from the datastore. This summary is a fairly general breakdown that can support a fairly wide range of reports related

to the existing reports. Thus, most changes to the reports do not require modifying the history; just its interpretation and tabulation. The history file is currently a few hundred MB in size and grows slowly over time. The second are summaries, in an intermediate report format (IRF), of tagsets still in the datastore. These latter summaries are never stored for any length of time; they are simply intermediate results for the daily update/report cycle.

More details on the report process are given below, but eventually, reports are generated. After the reports are generated, they are shipped to a web site for access by our customers.

**3.4 Process architecture:** The current processing architecture is fairly straightforward. With the exception of the first step, which occurs throughout the day, the remaining steps occur as part of the daily update cycle.

The first step is to distribute the incoming tags out to the datastore. For every source file, files with the same name are created in each filesystem (and not partition) and filled with tags that hash to any partition on that filesystem.

The next step examines all the filesystems, determines which input files will be included in this cycle, and then generates all the jobs to be executed for this update cycle. The 10,000+ jobs are then given to *woomera* for execution.

Within each filesystem, the incoming tags are sorted together in 1GB parcels. The resulting files are merged and then split out into each of the 52 partitions. The end result is the *add file*, a sorted file of tags to be added to each partition's data file. (The 1GB size is an administrative convenience; from this we experimentally tuned the various sorting parameters, most noticeably the amount of memory used.)

The next process, *pu*, updates the partition data file by merging in the new tags, deleting appropriate tagsets, and generating an IRF output for the new partition data file. The deleted tagsets are put into the *delete file*. We then generate an IRF output for the delete file. Because the underlying filesystem is unbuffered, all tag-related I/O goes through a *n*-buffering scheme (we currently use triple buffers). We generate a summary description

of the delete file.

The next step, performed after all the partitions on a specific filesystem have been processed, rolls up the two different report summaries in all those partitions into two equivalent files for the whole filesystem.

The next step generates the reports for that cycle. First, we combine the summary for the deleted tagsets with the old history file and generate a new history file. Second, we combine that and the 93 filesystem summaries and generate a single set of reports.

Finally, we backup the datastore in two passes. The first pass stores all the add files and delete files. The second pass stores a rotating sixth of the 4836 datastore partitions. (This is exactly analogous to incremental and full backups.)

**3.5 Numerology:** The system layout described above contains several seemingly arbitrary numbers; this section explains their derivation.

The number of filesystems (93) is a consequence of our RAID hardware. We had 16 cabinets, each with 2 UltraSCSI channels and arranged as 7 (35GB) LUNs. We wanted all our LUNs to have good but predictable performance so we used one LUN per filesystem and 3 LUNs per SCSI bus, giving us 96 possible filesystems. Three of these were needed for other purposes, leaving us 93 for the datastore. The 7th LUN in each cabinet was used for storage, such as the tag cache, not used during the daily update cycle.

The other numbers derived from a single parameter, namely how long it takes to process the average database partition. Although of little overall consequence, this affects how much work is at risk in a system crash and how long it takes to recover or reprocess a partition. We set this at 3 minutes, which implies an average partition size of 5-600MB. For a database of 2.6TB, this is about 5000 partitions. Here's where it gets weird: midway through Gecko's production life, we had to move the hardware from Mesa (Arizona) to Kansas City. It was infeasible to suspend the data feeds during the move, so we sent some RAID cabinets ahead to hold the data, which meant we had to run with only 78 filesystems for a few weeks, but after the move we would be back to 93. Changing the number of partitions is extremely

expensive (about 4 days clock time), but if the number of partitions is constant, then redistributing them amongst different numbers of filesystems (essentially renaming them), is relatively cheap (5-6 hours clock time for file copies). The least common multiple of 78 and 93 is 2418 and so we chose 4836 partitions, which meant 52 partitions per filesystem.

We chose 1GB for the parcel size as a compromise between two factors: it is more efficient to sort larger files, but larger files require much more temporary space. This latter restriction was critical; peak disk usage for sorting 1GB is 3GB which was 10% of our filesystem. Our average disk utilisation was about 85%, and thus we couldn't afford larger files.

#### 4. Current performance

We can characterise Gecko's performance by two measures. The first is how long it takes to achieve the report and cycle end gates. The second is how fast we can scan the datastore performing an ad hoc search/extract.

The report gate is reached when the datastore has been completely updated and we've generated the final reports. Over the last 12 cycles, the report gate ranged between 6.1 and 9.9 wall clock hours, with an average time of 7.6 hours. The cycle end gate is reached after the updated datastore has been backed up and any other housekeeping chores have been completed. Over the last 12 cycles, the cycle end gate ranged between 11.1 and 15.1 wall clock hours, with an average time of 11.5 hours. Both these averages comfortably beat the original requirements.

There are a few ways of measuring how fast we can scan the datastore. The first is *tagstat*, which is a C program gathering various statistics about the datastore. The second and third are different queries to a SQL-like selection engine (*comb*). The second is a null query 1, which is always true. The third is a simple query which selects tagsets with a specific originating number and biller: `(onum == 7325551212) && (bsid == 34)`. The speeds and total run times are

query	run time	speed
<i>tagstat</i>	71 min	606MB/s
null	110 min	392MB/s
simple	170 min	255MB/s

Note that we have not yet tuned *comb* to increase its performance.

## 5. Tools

The implementation of Gecko relies heavily on a modest number of tools in the implementation of its processing and the management of that processing. Nearly all of these have application beyond Gecko and so we describe them here. Most of the code is written in C and *ksh*; the remainder is in *awk*.

**5.1 Reliable file transmission:** By internal fiat, we used CONNECT:Direct for file transmission; it is essentially a baroque embellishment of *ftp*. No matter what file transfer mechanism we used, we wanted to avoid manual intervention in the case of file transfer errors. The scheme used is a very simple, very reliable one:

- a) the sender registers a file to go.
- b) the xmit daemon transmits the file, and a small control file containing the name, length and checksum, and logs the file as sent.
- c) on the receiving system, the rcv daemon waits for control files and upon receipt, verifies the length and checksum. If they match, the file is distributed (in whatever way is convenient) and is logged as a received file.
- d) after the xmit daemon has been idle for some time (typically 30 minutes), it sends a control file listing the last several thousand files transmitted.
- e) when the rcv daemon receives the control file list from d), it compares that list against its own and sends retransmit requests for any that appear to have been transmitted but were not received.

This has proved very resilient against all sorts of failures, including system crashes and cases where due to operator error, the file is safely received but inadvertently removed or corrupted (in this latter case, we can simply rerequest the file!).

**5.2 Parser generation:** One of the greatest

challenges for Gecko is being able to parse the various data tap feeds. Roughly 60% of our input are AMA records (standard telecommunications formats). There are about 230 AMA formats; these formats rarely change, but a few new formats are added each year. The rest are described by *copybooks*, which are the COBOL equivalent of a C struct definition combined with the *printf* format used to print it out. We have a reference library of tens of thousands of copybooks, but the data we need to parse seems to only involve a few tens of copybooks. The copybooks change fairly frequently.

In both cases, parsing has the same structure: we convert the raw record into an internal C structure, and then populate a tag from that C structure. The latter function has to be handcoded as it involves semantic analysis of the raw record fields. The former function is completely automated and depends only on either an online database of AMA record formats or the copybooks themselves. For example, we have a copybook compiler (a rather complicated *awk* script) that takes a copybook as input and produces C definitions for the equivalent structs, a procedure that takes an EBCDIC byte stream and populates the C equivalent (in ASCII where appropriate), and a structured pretty-printing function that lets you look at the EBCDIC byte stream in a useful way. Most copybook changes involve adding new members or rearranging members; this is handled transparently by just dropping in the new copybook. Over 80% of the C code in Gecko is generated in this fashion.

**5.3 Job management:** Gecko performs many thousands of jobs per day. We needed a tool that could support conditional execution (stalling some jobs when other jobs failed), sequential execution (stalling some jobs until other jobs have run), parallel execution (execute as many jobs in parallel as we can), and management of these jobs. The resulting tool, a job dispatch daemon, was called *woomera* (an Aboriginal term for an implement to enhance spear throwing), and *wreq* (which submits requests to *woomera*). The essential aspects of *woomera* are:

- the two main concepts are *jobs* and *resources*.
- jobs consist of a name, priority,



optional *after* clauses (or more accurately, prerequisite conditions), an optional list of resources and a ksh command.

- resources are capitalised names and have one major property: an upper limit.
- jobs become *runnable* when their prerequisites complete. A job prerequisite means waiting for that job to complete successfully (that is, with a zero exit status). A resource prerequisite means waiting for all jobs using that resource to finish successfully.
- when a job is runnable, it is executed subject to resource limits (and certain other limits, such as total number of jobs and actual machine load). If a resource *Rsrc* has a limit of 3, then at most 3 jobs using the resource *Rsrc* can run simultaneously.
- there are various administrative functions such as deleting jobs, dumping the internal state, and forcing a job to be runnable.

The ubiquitous use of *woomera* has been of enormous benefit. It provides a uniform environment for the execution of Gecko jobs, logging job executions, and a very flexible mechanism for controlling job execution. For example, during the daily datastore update, we need to halt parsing activities. This is simply done by making all the parse jobs use a specific resource, say *PRS\_LIMIT*, and setting its limit to zero. The jobs themselves are unaware of these activities.

**5.4 Execution management:** Initially, we controlled the job flow by manually setting various resources within *woomera*. After a while, this became mechanical in nature so we automated it as a ksh script called *bludge*. Every few minutes, *bludge* analyses the system activity and determines what state the machine is in, and sets various limits accordingly. For example, when the tag cache becomes uncomfortably full, *bludge* sets the limit for *PRS\_LIMIT* to zero so that no more tags will be produced and avoid the situation where tags would be thrown away. (Even though its a cache, recreating the data would likely involve accessing magnetic tape which we would really like to avoid!)

*Bludge* calculates the system state from scratch each times it runs, rather than "knowing" what ought to be happening or remembering what was happening last time. Although this is less efficient, it is far more robust and is resilient against ad hoc changes

in the system environment or workload.

**5.5 Tape store/restore:** Gecko has relatively simple needs for tape operations. There are three classes of files we backup to tape:

- raw tap files, backed up 6 times a day, retention is forever, about 40GB/day.
- full datastore backups (the actual data files), once per cycle, retention is 3 copies, about 400GB/day.
- incremental datastore backups (the add and delete files), once per cycle, retention is 3-6 months, about 135GB/day.

In each case, we have an exact list of the absolute filenames to backup. Recovery is infrequent, but it also uses a list of absolute filenames.

By internal fiat we were forced to use a specific product, Alexandria. We've been assured that someone is happy with Alexandria, but we are not. We've had to build and maintain our own database of files we've backed up in order to get plausible performance out of Alexandria. Overall, our simple store/restore (a list of files) operations require over 2000 lines of ksh scripts and about 1GB of databases as wrappers around the basic Alexandria commands.

In retrospect, the tape subsystem, consisting of the StorageTek Powderhorn silo with Redwood drives and Alexandria software, was surprisingly unreliable. Of the 600 tapes we wrote successfully, about 20 physically failed upon reading (tape snapping or creasing). Alternatively, about 15% of file recoveries failed for software reasons and would eventually succeed after prodding Alexandria (and sometimes, the tape silo) in various ways.

**5.6 Gre:** The Gecko scripts make extensive use of *grep*, and in particular, *fgrep* for searching for many fixed strings in a file. Solaris's *fgrep* has an unacceptably low limit on the number of strings (we routinely search for 5-6000 strings, and sometimes 20000 or so). The XPG4 version has much higher limits, but runs unacceptably slowly with large lists. We finally switched to *gre*, developed by Andrew Hume in 1986. For our larger lists, it runs about 200 times faster, cutting run times from 45 minutes down to 15 seconds or so. While we have not measured it, we would expect the GNU *grep* to perform about as well as *gre*. Both tools use variants of the



Commentz-Walter algorithm [Com79], which is best described in [Aho90] (the original paper has a number of errors). Commentz-Walter is effectively the Aho-Corasick algorithm used in the original *fgrep* program combined with the Boyer-Moore algorithm.

## 6. What we learned along the way

**6.1 Decouple what from how:** The natural way to perform the daily update cycle is to have some program take some description of the work and figure out what to do and then do it, much like the Unix tool *make*. We deliberately rejected this scheme in favour of a three part scheme: one program figures out what has to be done (*dsum*), and then gives it to another to schedule and execute (*woomera*), while another program monitors things and tweaks various *woomera* controls (*bludge*). Although superficially more complicated, each component is much simpler to build and maintain and allows reuse by other parts of Gecko. More importantly, it allows real-time adjustments (e.g. pause all work momentarily) as well as structural constraints (e.g. keep the system load below 60 or no more than 2 jobs running on filesystem *gb*).

The decision to execute everything through *woomera* and manage this by *bludge* has worked out extremely well. We get logging, flexible control, and almost complete independence of the mechanics of job execution and the management of that execution. We can't imagine any real nontrivial production system not using similar schemes. In addition, this has allowed us to experiment with quite sophisticated I/O management schemes; for example, without affecting any other aspect of the daily update cycle, we played with:

- minimising head contention by allowing only one job per filesystem (by adding a per filesystem resource)
- managing RAID controller load by restricting the number of jobs using filesystems associated with specific RAID controllers (by adding a per controller resource)
- managing SCSI bus load (by adding a per SCSI bus resource)

We typically conducted these experiments during production runs, manually adjusting limits and measuring changes in processing

rates and various metrics reported by the standard Unix system performance tools *iostat*, *vmstat*, and *mpstat*.

**6.2 Cycle management:** As described above, *bludge* manages the overall system environment by tweaking various resource limits within *woomera*. For example, when we are in the CPU-intensive part of the update cycle, *bludge* sets the limit for *PRS\_LIMIT* to zero in order to prevent tap file parsing during the update cycle.

More importantly for the cycle, *bludge* ensures that processing on all filesystems finishes at about the same time, thus minimising the overall cycle length (there is significant variation in the processing time required for each filesystem, and by simple round-robin scheduling, the cycle would take as long as the longest filesystem). Recall that the datastore update jobs have a resource indicating the filesystem containing the partition, say *LVOLgb*. Typically, we run about 50 update jobs simultaneously. So if *bludge* notices that filesystem *gb* is 70% done and filesystem *bf* is only 45% done, it will likely set *LVOLgb* to zero and *LVOLbf* to 1 or 2 until filesystem *bf* catches up.

**6.3 Recovery:** For the first few months of production, we averaged a system crash every 2-3 days. This caused us to quickly develop and test effective techniques for restarting our update cycle. The two key concepts were careful logging of program start and end, and arranging that programs like *pu* were transactions that either completed cleanly, or could be rerun safely (regardless if they had either failed or just hadn't finished).

**6.4 Centralising tag I/O:** All tag I/O flows through one module. While this seems an obvious thing to do, it has meant this module is the most difficult piece of code in the entire project, and for performance reasons, the most sensitive to code and/or operating system changes. The most visible benefits have been: performance improvements (such as when we changed from normal synchronous I/O to asynchronous multibuffered I/O) are immediately available to all tools processing tags or tagsets, application ubiquity (files can be transparently interpreted as files of tags or tagsets regardless of what was in the original file), and functional enhancements (such as when we supported internal tagset

compression) are immediately available to all tools processing tags or tagsets.

#### 6.5 Weakness of system hardware/software:

While most people would agree that Gecko is pushing the limits of what systems can deliver, we were surprised by how many system hardware and software problems impacted our production system. Most were a surprise to us, so we'll list a few as a warning to others:

- we originally had fewer, larger filesystems made by striping together 3 36GB LUNs. We expected to get faster throughput, but instead ran into controller throughput bottlenecks and baffling (to both Sun and us) performance results as we varied the stripe width.

- trying to force several hundred MB/s of sequential I/O through the page cache never really worked; it either ran slowly or crashed the system. Apparently, the case of sequentially reading through terabytes of disk was never thought of by the designers of the virtual memory/page cache code. (To be fair, large sequential I/O also seems to confuse system configurers and RAID vendors, who all believe more cache memory will solve this problem.) Tuning various page cache parameters helped a little, but in the end, we just gave up and made the filesystems unbuffered and put double-buffering into our application. (Of course, that didn't help the backup software or any other programs that run on those filesystems, but *c'est la vie*.)

- we ran into unexplained bottlenecks in the throughput performance of pipes.

- we ran into annoying filesystem bugs (such as reading through a directory not returning all the files in that directory) and features (such as the internal filename lookup cache has a hard coded name length limit; unfortunately all our source filenames, about 60-70 characters long, are longer than that limit!).

- it is fairly easy to make the Solaris virtual memory system go unstable when you have less physical swap space than physical memory. While this is an easy thing to avoid, it took several months before we found someone at Sun who knew this.

6.6 *Trust but verify*: A significant aspect of our implementation, and one we didn't anticipate, involves performing integrity checks

whenever we can. This extends from checking that when we sort several hundred files together, the size of the output equals the sum of the sizes of the input files, to whenever we process tagset files, we verify the format and data consistency. (And recently, in order to track down a bug in our RAID systems where a bad sector is recorded every 30-40TB, we have been checksumming every 256KB block of tag data we write and verifying the checksum after we close the file!) Although this is tedious and modestly expensive, it has been necessary given the number of bugs in the underlying software and hardware.

6.7 *Sorting*: the initial sorting takes about 25% of the report gate time budget. The original scheme split the source tag files directly into each partition, and then sorted the files within each partition as part of the *pu* process step. This ran into a filename lookup bottleneck. Not only did it require 52 times as many filename lookups (once per partition rather than once per filesystem), these lookups were not cached as the filenames were too long. The current scheme is much better, but we thought of a superior scheme, derived from an idea suggested by Ze-Wei Chen, but have not yet implemented it yet. Here, we would split the original source tag files into several buckets (based on ranges of the sorting key) in each filesystem. After we sort each bucket, we can simply split the result out to the partitions appending to the add file. This eliminates the final merge pass and avoids the pipe performance bottleneck.

6.8 *Distributed design*: The distributed layout of the datastore has worked out very well. It allows a high degree of parallel processing without imposing a great load on the operating system.

Although we have not yet made use of it, it also allows processing distributed across distinct systems as well as filesystems. If we had implemented Gecko on a central server and a number of smaller servers (rather than one big SMP), then the only significant traffic between servers would be the background splitting of tags out to the smaller server throughout the day and copying the rolled up report summaries back to the central server. This latter amounts to only a couple of GB, which is easily handled by modern LANs.

#### 6.9 Move transactions outside the database:

Because we only update the datastore as part of a scheduled process, we can assure the atomicity of that update operationally, rather than rely on mechanisms within the datastore itself. This had several advantages, including simpler datastore code, more efficient updates, and a simple way of labelling the state of the datastore (namely, the name of the update cycle performing the updates). This label was embedded in all the add and delete files, figured prominently in all the reports, and allowed complete unambiguity and reproducibility of both datastore and reports.

6.10 *Processing MVS feeds:* Although the most obvious problem in dealing with MVS feeds has been a surprisingly large number of file header/trailer sequencing schemes, the worst problem has been a simple one: the absence of a unambiguous date and time stamp. Some feeds only have a date, and not a time of day. But even those that do have a time of day neglect to indicate a timezone. It is quite hard, therefore, to nail down in absolute terms what time the file was generated. (We guess based on the processing center.)

6.11 *Tools, not objects:* Contrary to popular trends, our approach was very much tool based, rather than object based. It seems that this is a performance issue; if you really need a process to go fast, you make a tool to implement that process and tune the heck out of it (you don't start with objects and methods and so on).

6.12 *Focus:* We did one thing that really helped our design, which unfortunately might not be applicable to most developers: we had a clear vision of what our design could do well, and we rejected suggestions that did not suit the design. This sounds worse than it is; we are able to produce all the reports that our customers have required. However, as in all architectures, the customer will ask for things that seriously compromise the basic system design, and we denied those rather than warp or bloat the design.

### 7. Performance comparisons

The Gecko system is quite portable, requiring a regular ISO C environment augmented by sockets and POSIX threads. This allows to port the system to different systems

and do true benchmarking. This section will describe the results and price/performance for the original Sun system and an SGI system. We had intended to include a Compaq 4 CPU server but were stymied by inadequacies in PC environments easily available to us. (None of the POSIX environments for Windows, such as UWIN, support threads, and we have not yet been able to bludgeon the GCC/Linux environment and their so-called extensions into submission.) Given the poor compatibility of the various thread libraries, the only practical solution is to remove threads and depend on file system buffering to work.

We can calculate the price/performance rating for a system by combining three factors:

- 1) CPU speed (how fast can a single process do a specific amount of work)
- 2) system efficiency (how fast can the system execute a set of processes)
- 3) price

The overall rating is simply the product of these three numbers. Given the vagaries of computer pricing, this section will omit the pricing factor.

In more detail, we benchmarked the update cycle. (The other significant activity we do is parsing of tap files, and this is heavily CPU-bound and covered by 1) above.) The production task is to run 4836 *pu* jobs; our benchmark ran a smaller number that depended on the particular system capacity. It is infeasible to carry around terabytes of data for benchmarking. Our solution was to replicate a small group of filesystems to whatever size we need; we tracked total filesystem processing times and found *ja* was consistently around the 25th percentile, *ea* around the 50th percentile, and *nd* around the 75th percentile (we named our 93 filesystems as *[a-o][a-f]* and *p[a-c]*). To follow the real data sizes, we replicated groups of 5: *ja ea ea ea nd*.

We measured the CPU speed by averaging the user time needed to process all unique data files in the benchmark.



Task	Resource	Cur. Value	Comments
a1	networking	250GB/day	raw data feeds into the satellites
a2	CPU	760Ks/day	compress raw data feeds
a3	networking	40GB/day	compressed data feeds from satellites to <i>goldeye</i>
a4	CPU	265Ks/day	tag creation (from data feeds) into tag cache
a5	interFS	100GB/day	tags from tag cache to datastore filesystems
b1	CPU&FS	61H	sorting new tags
b2	CPU	166H	updating the datastore
b3	interFS	2GB/day	intermediate report files from datastore filesystems to <i>goldeye</i>
b4	CPU	0.5H	final report generation
b5	TAPE&FS	4H	datastore backup

Table 1: Performance model

Efficiency is simply the ratio of total CPU time to real (clock) time. The final factors are:

Factor	Sun E10k	SGI O2K	SGI/Sun
CPU	63.2s	49.0s	1.28
Efficiency	0.183	0.357	1.95

In this case, multiplying these factors together gives about a 2.5 advantage to SGI. Of course, performance is but one factor in choosing a vendor; in our case, Sun was chosen for other reasons. In addition, your application will behave differently. But do keep in mind the issue of system efficiency; the size of this factor was a surprise to us.

## 8. Scalability

We are often asked whether Gecko is scalable; the answer is "Of course!" The real question is somewhat different: given a specified workload, do we have a way to predict the expected processing time and the necessary resources? Yes, we do.

The model, shown in Table 1, concentrates on the two main aspects of any batch-oriented system; moving bytes, and processing bytes. In our discussion, we'll refer to two architectures: one is the SMP scheme that we described above, and the second is a (smaller) central server connected to a cluster of small machines with the datastore distributed amongs the small machines. In the following, we will denote the amount of incoming tags for a day by  $i$ ; the size of the datastore by  $d$ , and the total amount of tape throughput is  $t$ .

The first section is the work that runs

asynchronously from the daily update cycle. Resources consumed by tasks a1, a2 and a3 are linear in the size of the raw data feeds. We can increase the resources for a1 and a2 at a linear rate by simply adding more satellites. The networking for a3 will scale almost linearly until we hit fairly large limits imposed by hardware limits on the number of network cards; this is unlikely to be a problem as modern large servers can support several 100BaseT or GigaBit connections capable of supporting 250+GB/hour. For the cluster architecture, we could implement the loading dock as a separate system and start replicating that. The CPU needed for a4 will scale linearly. Task a5 on *goldeye* (SMP) takes about 2 hours; perversely, it would probably run faster over a network to a cluster. It should scale linearly until we run into either backplane limits on total I/O movement or operating system bottlenecks (the two ones we've seen most are virtual memory related and file system related locking).

The second set of tasks ( $b?$ ) make up the daily cycle. The tasks b1, b2, b3 and b4 can run in parallel limited only by the number of processors and independent filesystems available. In particular, tasks b1 and b2 are composed of 4836 subtasks, all of which can be run in parallel. Task b1 involves sorting and thus takes  $O(i \log i)$ ; b2 and b3 are both  $O(d)$ . Task b4 takes time linear in sum of the number of days represented by data in the datastore and the history file. Task b5 will take  $O(\frac{i+d}{t})$ .

Our Sun implementation would probably process up to 2-3 times the daily input



that we do now. Switching to a faster SMP, such as an SGI, could push that to 4-6 times. For real growth, though, you would want to go with a cluster implementation. Carefully constructed, this would make nearly every aspect of the workload scale linearly by adding more systems. The only task that doesn't is the sorting step b1, and even that could be mitigated by presorting files as they arrive on each filesystem and thus make the part of b1 necessary for the update cycle be a linear performance merge pass.

## 9. Conclusion

By any measure, Gecko aims at solving a very large problem. Indeed, originally it was thought that the problem was not solvable at all (and during our darkest days, we almost believed this as well). But the fact remains that we have a system in production today that handles the volumes and meets its deadlines. Furthermore, the project initially went live only 8 months after starting with all design and development done by a team of 6 people.

And even after a year, the volumes are still stunning. On an average day, we process about 240GB of legacy data, add about 1B tags to a 13B tagset datastore stored on 2.6TB of disk, and backup about 900GB of data to tape. And on our peak day (recovering from our system's move from Mesa to Kansas City), those numbers have stretched to 5B tags added and 1.2TB backed up to tape.

The datastore design, a myriad of sorted flat files, has proved to be a good one, even though it isn't a conventional database. It works, it comfortably beats its processing deadlines, and has proved flexible enough to cope with several redesigns.

## References

- [Aho90]Alfred V. Aho, *Algorithms for Finding Patterns in Strings*, Handbook of Theoretical Computer Science, Elsevier, 1990. pp 278-282.
- [Com79]Beate Commentz-Walter, *A string matching algorithm fast on the average*, Proceedings of the 6th Internat. Coll. on Automata, Languages and Programming, Springer, Berlin, 1979. pp 118-132.
- [Gre99]R. Greer, *Daytona and the Fourth-*

*Generation Language Cymbal*, ACM SIGMOD Conference, June 1999.

- [Hum99]A. Hume and A. MacLellan, *Project Gecko: pushing the envelope*, NordU'99 Proceedings, 1999.
- [Kor86]H. F. Korth and A. Silberschatz, *Database System Concepts*, McGraw-Hill, 1986.
- [Lau79]Hugh C. Lauer and Roger M. Needham, "On the Duality of Operating System Structures", *Operating Systems Review*, 13(2), 3-19 (1979).

## Acknowledgements

This work was a team effort; the other Gecko developers are Ray Bogle, Chuck Francis, Jon Hunt, Pam Martin, and Connie Smith. There have been many others within the Consumer Billing and Research organisations with AT&T who have helped; in fact, too many to list here. We have had invaluable help from our vendors, but Martin Canoy, Jim Mauro and Richard McDougall from Sun were outstanding.

The comments of the reviewers and our shepherd greatly improved this paper; the remaining errors are those of the authors. We thank Rob Kolstad for suggesting the Lauer and Needham paper.

This is an experience paper, and as such, contains various statements about certain products and their behaviour. Such products evolve over time, and any specific observation we made may well be invalid by the time you read this paper. Caveat emptor.



# Extended Data Formatting Using Sfmt

Glenn S. Fowler, David G. Korn and Kiem-Phong Vo  
AT&T Laboratories – Research  
180 Park Avenue, Florham Park, NJ 07932, U.S.A.  
gsf,dgk,kpv@research.att.com

## Abstract

*The ANSI-C Standard defines Stdio as the I/O library for C programs. Despite its ubiquitous use, Stdio has well-documented deficiencies in various areas including data formatting. The Sfmt library provides an alternative to Stdio with improved functionality, robustness and performance. In particular, Sfmt extends the data formatting functions so that applications can deal with arbitrary scalar objects, avoid unsafe operations and even define their own conversion patterns. This paper discusses these formatting enhancements.*

## 1 Introduction

The Stdio `printf()`/`scanf()` family of functions [1, 5] are the de facto standard for formatting data in C programs. Many implementations of the C++ I/O operators [9] `>>` and `<<` are also based on the `printf()`/`scanf()` functions. Despite this popularity, the Stdio formatting functions have a number of shortcomings:

- *Inadequate handling of abstract scalars:* The formatting functions only deal with primitive scalar types. To format an abstractly defined scalar object, it is customary and even necessary to cast it to some presumed larger scalar type. For example, on most platforms, a file offset object declared with the Posix type `off_t` would need to be casted to a `long` for printing. This trick is not portable since, on a modern platform, `off_t` may be defined on top of a newer and larger type such as `long long`.
- *Unsafe data scanning:* String scanning with Stdio always runs the risk of overflowing the buffer because there is no way to tell the scanning function the buffer size. Buffer overflow bugs often corrupt memory leading to disastrous consequences. These bugs are also hard to detect.
- *Inextensible interface:* It is useful to be able to extend the defined set of conversion patterns or even to redefine some of them based on specific needs. For example, if an application defines

a type `Coord_t` for spatial coordinates, it would be nice to be able to define a corresponding formatting pattern, say `%C`, to print or scan such a type. This cannot be done in the current formatting framework.

- *Inadequate reuse:* The POSIX Standard [8] defines commands such as `printf` to format data in the same style as the corresponding Stdio functions. Since applications cannot access the format parsing and argument processing code in the formatting functions, each tool must invent its own way to perform these tasks. This unnecessarily duplicates work already done in library functions and does not help to improve interface consistency across tools.

The Sfmt library [3, 7] was introduced in 1991 as a better alternative to Stdio. In particular, the Sfmt data formatting functions outperformed their Stdio counterparts due to faster integer and floating point value conversion algorithms. Although these early Sfmt formatting functions addressed the mentioned portability and robustness issues in Stdio, they were still inflexible so that applications could not adapt them for specific needs.

Starting from the 1997 release of Sfmt, we experimented with extending the formatting functions to allow both non-standard patterns and alternative argument processing. The early extensions were useful but we found through experience that the framework was incomplete and cumbersome to use. For example, formatting flags and values such as width and precision were not properly packaged and passed

between library and application code when processing non-standard conversion patterns. It was also impossible to redefine existing conversion patterns. Since then, we have redesigned the extensions to enable much more natural cooperation between the formatting functions and applications.

The rest of this paper summarizes the new formatting features and gives examples of how to use them. A performance study comparing Sflo and various Stdio versions on the basic printing and scanning tasks shows that Sflo outperforms Stdio despite the additional features.

## 2 Extended data formatting

The formatting extensions include portable scalar formatting, safe data scanning, dealing with integers in general bases, and the ability to define new formatting patterns or redefine existing ones. To accommodate the new extensions, the general forms of Sflo printing and scanning patterns are respectively:

```
%[pos$][flag][(tstr)][width[.precis[.base]]]z
%[*][pos$][flag][(tstr)][width[.width.base]]z
```

Arguments such as `pos$`, `width`, etc. are the same as defined in the ANSI-C Standard. The `base` argument is introduced to accommodate generalized scalar and string processing.

The argument `(tstr)` is used to define a string that will be passed to an extension function if one is defined. Section 2.5 discusses how applications can use such data for non-standard conversion patterns and argument processing.

The below subsections discuss the new extensions. We mostly present printing examples, but scanning examples work in a similar way.

### 2.1 Portable scalar formatting

Certain platforms provide 64-bit integer and floating point values via types such as `long long` and `long double`. These types are handled differently in different Stdio implementations. For example, the Microsoft-C version provides an `I64` flag to specify a 64-bit integer while other Unix platforms use the more general flag `ll` for the same purpose.

Sflo generalizes the `ll` flag to deal with the largest primitive types on a particular platform. In fact, to ensure portability, Sflo provides types such as `Sflong_t`, `Sflong_t` or `Sfdouble_t` that are always mapped to the largest primitive types available. The following examples show how to use the `ll` flag in printing or scanning objects with large types:

```
Sflong_t  intval;
sprintf(sfstdout,"%lld", intval);

Sfdouble_t fltval;
sfscanf(sfstdin,"%llf", &fltval);
```

The `ll` flag enables printing of abstract types that may be mapped to different primitive scalar types on different platforms. For example, the familiar ANSI-C `size_t` for memory size and the POSIX `off_t` for file offset are often mapped to `unsigned int` and `long` respectively. But `off_t` may also be mapped to the type `long long` on platforms that support very large files. To print a value defined by an abstract scalar type, one should cast it to the largest corresponding scalar type and use the `ll` flag with an appropriate conversion pattern. For example, an `off_t` value should be printed by casting to `Sflong_t` and using the pattern `%lld` as follows:

```
off_t  offset;
sprintf(sfstdout,"%lld", (Sflong_t)offset);
```

Unfortunately, the above trick does not work with scanning since the scanned value must be stored in a location with a specific type. Printing performance is also suboptimal if arithmetic operations on such large types are more expensive than that on normal types. Various proposals are being debated by the C9X Standard Committee [4] to solve this problem. For Sflo, since we already needed to provide the Microsoft-C flag `I64` for portability, we simply took the opportunity and generalized this flag to allow specification of objects with arbitrary sizes. The below examples show how this works:

```
sprintf(sfstdin,"%I4d",intval);
sprintf(sfstdin,"%I*d",sizeof(intval),intval);
sfscanf(sfstdin,"%I*f",sizeof(fltval),&fltval);
sprintf(sfstdout,"%I64d",big_long);
```

The first line indicates that the integer value `intval` is an object whose size is 4 bytes, i.e., a 32-bit integer. The second line is more general and supplies the size of `intval` via `'*'`. This will work with integers of any size. The third line is similar to the second line but for scanning a floating point value. The fourth line shows that, for compatibility with Microsoft-C, the value 64 can be used to identify a 64-bit integer.

The above use of 64 to indicate bit size instead of byte size is potentially ambiguous. However, it will be a long time before we need to worry about machines with 64-byte words. In the mean time, it solves a practical problem.



## 2.2 Safe data scanning

The string scanning patterns `%s`, `%c` and `%[]` are often unsafe to use due to buffer overflow problems. The aforementioned `I` flag can be used to define buffer sizes. Specifying a buffer size does not limit the amount of scanned data. Rather, scanned data exceeding the buffer limit are discarded. Below are two scanning examples where the second one is slightly more general than the first:

```
char buf[10];
sfscanf(sfstdin,"%I10s",buf);
sfscanf(sfstdin,"%I*s",sizeof(buf),buf);
```

In both cases, at most 9 bytes will be copied into the buffer. Further input data will be scanned but discarded. `Sfio` reserves one byte from the buffer for the final null character.

## 2.3 Integers in general bases

The patterns `%i`, `%u` and `%d` can format in bases from 2 to 64. The syntax `[width[.precision[.base]]]` is used so that a base is defined if and only if exactly two dots have appeared. If a base is not validly defined, base 10 is used. Below are the 64 digits used to represent numbers:

```
0123456789
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ @_
```

Pattern	Value	Result
<code>%.2d</code>	123	1111011
<code>%#.2d</code>	123	2#1111011
<code>%#..16d</code>	12345	16#3039
<code>%#..34d</code>	-12345	-34#an3
<code>%#..63d</code>	123456789	63#7QKgA

Table 1: Integer values in general bases

Table 1 shows examples of printing numbers in general bases. The flag `#` outputs a number in the form `base#representation` where `base` is decimal and `representation` is in the digits for that base.

## 2.4 Character and string arrays

In addition to handling characters and strings, the string patterns `%c` and `%s` can also print null-terminated arrays of characters or null-terminated arrays of strings. To format an array, a separator must be supplied based on the syntax `[width[.precision[.separator]]]`. That is, a separator is defined if and only if exactly two dots have

appeared. When the separator is given in the format string, it must be a non-alphanumeric character and appear immediately before the conversion pattern. Each formatted character or string always obeys the layout rules defined by width and precision. Below are three example formatting calls and results:

```
fprintf(sfstdout,"%...c", "abc");
a:b:c

fprintf(sfstdout,"%6...s|", '|', words);
| trez| tres| three|

fprintf(sfstdout,"%...s", words);
treztresthree
```

The second and third examples assume that the null-terminated array `words` contain three words: `trez`, `tres` and `three`. In the second example, the field width for each word is 6 and the `'|'` means to get the separator from the argument list. In the third example, the separator is not defined so the strings are simply output one after another.

## 2.5 Extended format processing

Applications can both define new conversion patterns and redefine existing ones. In addition, it is also possible to use call-back functions to get the objects to be formatted instead of getting them from the function argument list. This ability is important for implementing certain Posix commands such as `printf` that mimics the `Stdio` function with the same name but whose arguments are given as strings on the command line.

In the below, we first describe the mechanisms to extend formatting, then give examples of how they may be used. Readers not yet acquainted with these extensions may prefer to reverse the order by reading the examples first before learning the details of the mechanisms.

### 2.5.1 Formatting environments and stacks

A typical formatting call has as input arguments a formatting string with conversion patterns and a corresponding argument list of the objects to be formatted based on the specified patterns. Such a formatting string and argument list is called a *formatting pair*.

To extend pattern processing, we define *formatting environments* in which both formatting pairs and associated call-back functions can be given. We further allow these multiple formatting environments to be stacked on top of one another on an *environment stack* for recursive pattern processing.

```

Sffmtext_f  extf;
Sffmtevent_f eventf;

char*      form;
va_list    args;

int         fmt;
ssize_t    size;
int         flags;
int         width;
int         precis;
int         base;

char*      t_str;
int        n_str;

```

Figure 1: The extended formatting environment

A formatting environment is of the type `Sffmt_t` with elements as shown in Figure 1:

- The first four members of an `Sffmt_t` object should be set by the application before passing to the formatting function. The event-handling function `eventf`, if not `NULL`, is called to process events such as popping the stack. `form` and `args` define a new formatting pair if `form` is not `NULL`. The extension function `extf`, if not `NULL`, is called to process conversion patterns.
- The next six members of `Sffmt_t` are used by the formatting function and `extf` to exchange data about the pattern being processed. For example, on the call to `extf`, the formatting function sets `fmt` to the pattern being processed. On return, `extf` may reset that field to redirect further processing.
- The last two members of `Sffmt_t` are used by a formatting function to pass to `extf` the (`tstr`) string that `Sfio` allows in specifying a conversion pattern. Section 2.5.5 shall discuss a use of such strings to unify formatting at the command level.

Each formatting call maintains a separate formatting stack whose bottom is a virtual formatting environment that consists only of the original formatting pair. A new conversion pattern `%!` is used to either push a new formatting environment onto the formatting stack or change the extension functions of the top environment. This works as follows:

- When the pattern `%!` is encountered during processing of a format string, the formatting

function obtains the corresponding `Sffmt_t` object. Then, if the `form` field of this object is not `NULL`, the new environment is pushed onto the stack and processing will start with the new formatting pair and extension functions. On the other hand, if `form` is `NULL`, only the extension functions of the current top environment are changed to the new ones and processing continues with the current formatting pair.

- The stack top is popped whenever its format string is completely processed or if a call to an extension function returns a negative value. When this happens, the current `eventf` function will be called to allow the application to perform any finalization actions (e.g., freeing the formatting environment object).
- To process a conversion pattern, the formatting function first fills the relevant `Sffmt_t` fields with data such as the current states of the format string and the argument list, the formatting pattern, object size, flags, width, precision, etc. Then, it makes the call `(*extf)(f,v,fe)`. Here `f` is the stream, `v` is a pointer to an object suitable for storing a scalar or pointer value, and `fe` is the given `Sffmt_t` object.
- The return value of `extf` is handled as follows:
  - A negative value pops the stack. Processing will continue with the newly revealed top environment if there is one. If there is no more environment, the formatting function will return.
  - A positive value means that `extf` has finished formatting this pattern and also indicates the amount of stream data that `extf` reads or writes. The calling formatting function will record this amount, then continue processing of the format string.
  - A zero value indicates that the formatting function should take over processing this pattern. The extension function may redirect processing by modifying the `Sffmt_t` object to change the formatting pattern and other associated formatting attributes. In fact, if the original pattern was not one already defined by `Sfio`, `extf` should reset the field `fmt` to one already defined. Otherwise, this conversion pattern will be ignored.

```

1. timeprint(Sfio_t* f,Void_t* v,Sffmt_t* env)
2. {   if(env->fmt == 't')
3.     {   time_t t = va_arg(env->args,time_t);
4.         *((char**)v) = ctime(&t);
5.         env->size = -1;
6.         env->fmt = 's';
7.         env->flags |= SFFMT_VALUE;
8.     }
9.     return 0;
10. }

11. error(char* form, ...)
12. {   Sffmt_t    fmt;
13.     va_list     args;
14.     static int count;

15.     va_start(args,form);
16.     fmt.form = form;
17.     va_copy(env.args,args);
18.     fmt.extf = timeprint;
19.     fmt.eventf = (Sffmtevent_f)0;

20.     sprintf(sfstderr,"Error #%d, %!.\\n",
              ++count, &fmt);

21.     va_end(args);
22. }

23. error("%t:\\n\\tTrying to allocate %d bytes",
        time(0), 1024);

```

```

Error #1, Tue Dec 1 00:39:46 EST 1999:
Trying to allocate 1024 bytes.

```

Figure 2: An error processing function

### 2.5.2 Defining a new pattern

Figure 2 shows how to implement a function `error()` that prints all normal conversion patterns and also supports a new pattern `%t` to convert a clock value to a date string. This example also shows how the formatting stack is used.

Lines 1-10 define an extension function `timeprint()` to interpret the new conversion pattern if it is specified. Other patterns are simply deferred to the calling formatting function.

Lines 3-4 obtain the time value and convert it to a date string. The use of `time_t` to get a value off of an argument list is possible here because `timeprint()` is an application routine. Both `time_t` and `ctime()` are defined in the ANSI-C Standard.

Lines 5-7 reset the formatting pattern `env->fmt` to `'s'` and `env->size` to `-1` and also add the bit flag `SFFMT_VALUE` to `env->flags`. These actions tell `sprintf()` that `timeprint()` is returning a null-

terminated string to be printed. Although not necessary in this example, the extension function should always make sure that associated formatting attributes such as width, precision and base are reset properly along with resetting a conversion pattern.

Lines 11-22 define a function `error()` to print error messages with embedded conversion patterns including `%t`.

Lines 15-19 construct a formatting environment `fmt` from the function arguments and the extension function `timeprint()`.

Line 17 shows the use of the macro function `va_copy` to copy argument lists. This macro function is provided by `Sfio` for portability.

Line 20 calls `sprintf()` to do the actual formatting. This call first outputs an error count. Then when it encounters the pattern `%!.`, it stacks `fmt` to start processing the arguments of `error()`. When that is finished, the stack is popped and processing returns to the original formatting string to output the final period.

Line 23 gives an example of how `error()` may be called to print an error message concerning an allocation error. The `%t` pattern is treated by `timeprint()` in the described manner. However, `timeprint()` simply returns 0 for the `%d` pattern so that `sprintf()` will continue with normal processing. An example output is shown after the `error()` call.

### 2.5.3 Redefining a pattern

Figure 3 shows an example that redefines the system-defined pattern `%c` and also defines a new pattern `%C` to print a pair of real numbers in two different ways, as a complex number or as a two-dimensional coordinate. The former is presented as a pair of numbers in parentheses while the latter is presented in angle brackets.

Lines 1-4 define the object type `Obj_t`, a struct with two floating point value members.

Lines 5-17 defines the extension function `objprint()` to print an `Obj_t` object based on the specified formatting patterns. The default clause of the `switch` statement shows that `objprint()` returns 0 on all conversion patterns other than `%c` and `%C`. This means that `sprintf()` will continue processing them normally.

Lines 8-13 show how recursive calls to `sprintf()` are used to process the patterns `%c` and `%C`. In each case, data is output directly to the stream. The output amount is returned to indicate to the the original `sprintf()` call that the pattern has been completely processed and also so that `sprintf()` can correctly update its output count.

```

1. typedef struct obj_s
2. { double   x;
3.   double   y;
4. } Obj_t;

5. objprint(Sfio_t* f, Void_t* v, Sffmt_t* env)
6. { Obj_t* o;
7.   switch(env->fmt)
8.   { case 'c': /* print a complex number */
9.     o = va_arg(env->args, Obj_t*);
10.    return sprintf(f, "(%g,%g)", o->x, o->y);
11.   case 'C': /* print a coordinate pair */
12.     o = va_arg(env->args, Obj_t*);
13.     return sprintf(f, "<%g,%g>", o->x, o->y);
14.   default :
15.     return 0;
16.   }
17. }

18. Sffmt_t fmt;
19. fmt.form = (char*)0;
20. fmt.extf = objprint;

21. Obj_t obj = {1.11, 2.22};

22. sprintf(sfstdout, "%!%c\n", &fmt, &obj);
    (1.11, 2.22)

23. sprintf(sfstdout, "%!%C\n", &fmt, &obj);
    <1.11, 2.22>

```

Figure 3: Printing user-defined data

Lines 18-20 construct a formatting environment. The field `fmt.form` is set to `NULL` so that only the extension function of the current top environment would be changed to `objprint()`.

Lines 21-23 initialize an object `obj` with the shown values, then print it both as a complex number and as a two-dimensional coordinate. The resulting outputs are shown along with the respective calls.

## 2.5.4 Application-defined arguments

Figure 4 shows how to extend `sfprintf()` so that the values to be formatted can be obtained either from the argument list or via a call-back function that gets them from the process environment.

Lines 1-18 define the function `envprint()` to process environment variables. The special processing is done only when an environment variable name is given via the use of the `(tstr)` syntax.

Lines 4-5 construct the name of the environment variable. This explicit construction is necessary because the `(tstr)` string `env->t_str` is not necessarily null-terminated.

```

1. envprint(Sfio_t* f, Void_t* arg, Sffmt_t* env)
2. { char name[1024], *v;

3.   if(env->n_str > 0)
4.   { memcpy(name, env->t_str, env->n_str);
5.     name[env->n_str] = 0;
6.     if((v = getenv(name)) && *v)
7.     { *((char**)arg) = v;
8.       env->size = -1;
9.       env->fmt = 's';
10.    }
11.   else
12.   { *((char*)arg) = '?';
13.     env->fmt = 'c';
14.   }
15.   env->flags |= SFFMT_VALUE;
16. }

17. return 0;
18. }

19. Sffmt_t ft;
20. ft.extf = envprint;
21. ft.form = (char*)0;

22. sprintf(sfstdout, "%!%s=%(*d)\n",
    &ft, "LINES", "LINES");
    LINES=24

23. sprintf(sfstdout, "%!%s=%(*s)\n",
    &ft, "SHELL", "SHELL");
    SHELL=/bin/ksh

24. sprintf(sfstdout, "%!%s=%(*s)\n",
    &ft, "UNKNOWN", "UNKNOWN");
    UNKNOWN=?

```

Figure 4: Application-defined arguments

Lines 6-15 attempts to obtain the value of the specified environment variable. If this value exists, it is returned in the given argument `arg`. The conversion pattern is changed to `'s'` since this is a string. If the value does not exist, the character `'?'` is returned and the conversion pattern is accordingly changed to `'c'`. In either case, the flag `SFFMT_VALUE` is set to indicate that further processing of the returned value is needed by the original `sfprintf()` call.

Lines 19-21 set up a new formatting environment. Since the field `form` is set to `NULL`, only the extension function of the current top formatting environment on the formatting stack will be changed.

Lines 22-24 give examples of printing the names and values of three environment variables: `LINES`, `SHELL` and `UNKNOWN`. In each case, the conversion pat-



tern `%!` is used to change the extension function to `envprint()`. After that, processing continues with the current formatting string and argument list. This would cause the name of the variable and the character `'=` to be output. Then, the `*` directive in the `"(tstr)"` construct obtains the second instance of the variable name from the argument list to pass on to `envprint()`. In turn, the `envprint()` call computes and returns the value of the specified environment variable in the manner described above.

### 2.5.5 Command-level formatting

Commands like `ls`, `ps` and `find` can produce data in tabular formats. Classic implementations provide a variable format controlled by option flags, each flag enabling another column in the formatted output. These commands have been independently extended by various groups to allow `printf`-style specifications, but because of the earlier lack of a programmable `printf` interface, such extensions are often incompatible.

The `Sfio` `"(tstr)"` construct allows a common syntax for extending formatting at command level. For example, our `ls` command provides a `-f format` option that accepts format parameters of the form:

`%[-+][width[.precis[.base]]](id[:subformat])char`

Here, `id` is path or any member of the `<sys/stat.h>` `stat` structure (with the leading `st_` omitted.) If `char` is `s` then the string representation of the item is formatted; otherwise, the integer form is formatted. Consider the below example option:

```
-f '%(mode)s %(mtime:time=%H:%M:%S)s %(path)s'
```

This would print:

- The file mode in the style of `ls -l`,
- The file modify time using the `strftime(3)` format `%H:%M:%S` (hours, minutes, seconds), and
- The file path name.

Within the `ls` implementation, such an option is simply passed to the formatting function `sfprintf()` after a formatting environment has been set up with an appropriate extension function that knows how to interpret the mentioned `(tstr)` strings. Then, `sfprintf()` parses the format string and calls the extension function for actual formatting.

Figure 5 shows parts of an extension function `lsprint()` to interpret the above example `(tstr)` strings for printing the path name and modification time of a file. Although this code is not the same as

```
1. typedef struct _lsfmt_s
2. {   Sffmt_t      fmt;
3.     struct stat* sb;
4.     char*        path;
5. } Lsfmt_t;

6. lsprint(Sfio_t* f, Void_t* arg, Sffmt_t* env)
7. {
8.     Lsfmt_t* ls = (Lsfmt_t*)env;

9.     if(...path name...)
10.    {   *((char**)arg) = ls->path;
11.        env->size = -1;
12.        env->flags |= SFFMT_VALUE;
13.        return 0;
14.    }
15.    else if(...modification time...)
16.    {   char buf[1024], pattern[1024];

17.        ...extract strftime() pattern...
18.        strftime(buf, sizeof(buf), pattern,
19.                localtime(ls->sb->st_mtime));

20.        return sfwrite(f, buf, strlen(buf));
21.    }
22.    ...
23. }
```

Figure 5: Printing file modification time

in our implementation of the `ls` command, it shows how the formatting extensions may be used.

Lines 1-5 define a type `Lsfmt_t` that combines the `Sffmt_t` type, a `struct stat*` for a file status object, and a `char*` for the file name. In this way, the `ls` application can pass along the file status data and file name to the formatting function. C casting rule allows a pointer to a `Lsfmt_t` object to be treated as a pointer to an `Sffmt_t` object. This way of extending a data structure to be passed back and forth between the library and the application code is commonly used in our libraries based on the discipline and method library architecture[10].

Line 9 identifies a formatting request for a path name via examining the string `env->t_str` to see if it defines the `id path`.

Lines 10-14 simply return the path name as a string to be further processed by the calling formatting function.

Line 16 identifies the print modification time request by examining the string `env->t_str` to see if it defines the `id mtime`.

Line 18-20 extracts from the `form` field the conversion string `%H:%M:%S` to pass to the ANSI-C function `strftime()`. This conversion string is assumed

to be stored in the buffer pattern. The function `localtime()` is called first to convert the `time_t` value `env->sb->st_mtime` to an object of the type `struct tm` as required by `strftime()`. Both `strftime()` and `localtime()` are defined in the ANSI-C Standard.

Line 21 writes the result out to the given stream and returns the number of bytes written. Subsequently, the formatting function continues with processing the format string.

### 3 Performance

The new features do add complexity to the formatting functions. Since many applications, especially those based on Stdio, only use the basic formatting tasks, we need to assure that their performance are not adversely affected by the new features when they are not used. Toward this end, we perform a study comparing Sfio against various Stdio versions on basic data printing and scanning tasks.

	Hardware	MHZ	OS
O	Pentium II	200	SCO UNIX 3.2
K	Pentium II	333	UWIN/WIN32
F	Pentium II	333	Linux 2.2.12-20
W	Pentium II	450	BSDI 4.0.1
D	HP9000/889	400	HP-UX B.10.20
G	UltraSparc2	2x300	SUNOS 5.6
R	SGI Origin 200	4x270	IRIX64 6.5
T	DEC-Alpha	500	UNIX V4.0D

Table 2: Tested platforms

Table 2 shows the platforms used in the performance study. The first four systems are PCs running various Unix operating systems. UWIN/WIN32 is David Korn's UWIN system [6] that provides a Posix layer on top of the WIN32 environment. The last four are large servers from Hewlett-Packard, Silicon Graphics, Sun Microsystems and Digital Equipment running some respective Unix operating systems available from the vendors.

The conditions of the experiments were as follows:

- A test program prints 25,000 lines out to a file, then scanning the same lines back. Each line contains an instance of each of the patterns: `c,d,o,x,f,e,s`. The amount of data generated per run is about 1.7Mbs.
- To ensure uniformity, we wrote a single test program based on the Stdio interface. To test Stdio on a particular platform, we simply compiled the program using the native `stdio.h` header

and library. To test Sfio, we compiled the program using the source compatibility header `stdio.h` provided by Sfio. This header mapped Stdio calls to Sfio calls mostly via macros. Such mappings did add some more work to the Sfio tests but we deemed that to be insignificant compared to the work done by the formatting tasks themselves.

- For UWIN/WIN32, the native WIN32 Stdio was used instead of the UWIN Stdio since the latter is just the same source compatibility interface provided by Sfio.
- The test programs always performed I/O to a same file in `/tmp`. In our environment, this ensured that the file would be on a disk local to the processing computer.
- All test runs were performed at night on lightly loaded machines. In fact, most machines were single user during the tests except for platform R, a large compute server.
- Times shown are totals of CPU and System time measured in seconds. Each data point was obtained as follows. Each test was run nine times. Then the highest two and lowest two values are discarded to remove certain outliers due to file caching effects on some platforms. The remaining five values are then averaged to produce the final result.

	Printing		Scanning	
	Sfio	Stdio	Sfio	Stdio
O	.82	1.01	.86	1.00
K	.42	1.96	.61	.73
F	.52	1.29	.66	1.22
W	.21	.43	.26	.30
D	.85	.90	2.06	2.07
G	.75	.85	.78	1.09
R	.40	.40	.49	.70
T	.25	.26	.33	1.39

Table 3: Timing results

Table 3 presents timing results on the mentioned platforms. Figure 6 shows the same data in bar charts. Below are a few comments on the data:

- Sfio outperforms Stdio on all platforms. Most of the improvement is due to new data conversion algorithms. For example, the decimal printing algorithm uses table look-up and an inline binary search to compute digits instead of

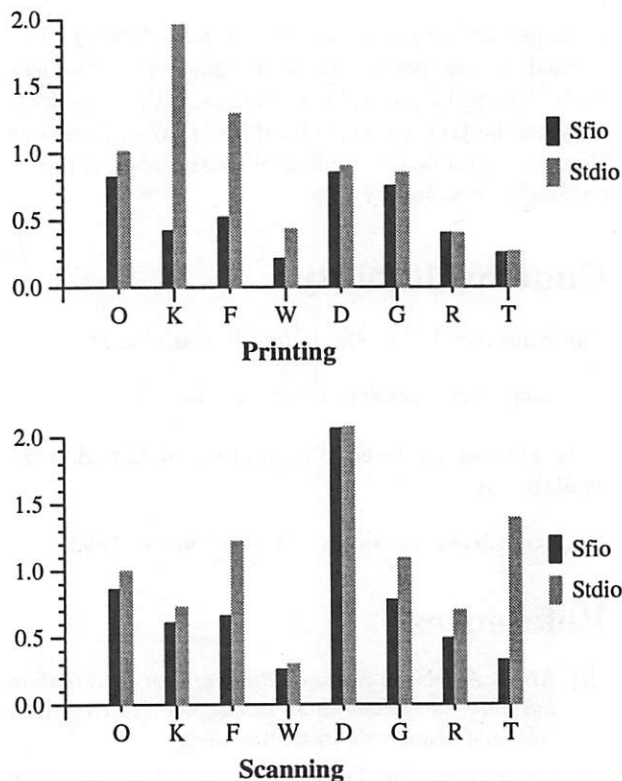


Figure 6: Formatting performance

the usual method of division and modulo by 10. This works particularly well on hardware such as SUN SPARCs that use function calls for division and modulo. A full description of the Sflo conversion algorithms is not appropriate for this paper whose focus is on the new formatting features. Interested readers can peruse the freely available Sflo source code for details.

- It should be noted that Sflo is built from a single source base but configured differently on different platforms. For optimal performance, it is important that certain basic functions, e.g., string or memory copy, are matched to the best available methods on a particular platform. We use the tool Iffe[2] to automatically detect and compare different functions made available by a platform for the same purposes and generate appropriate configuration parameters.
- The native Stdio on platform K, Windows NT, has the worst printing performance relative to Sflo. Some of this poor performance is due the buffering strategy of the I/O package (i.e., small stream buffer) but a larger part is due to antiquated conversion algorithms.

- Platforms K and F are based on the same processor but use different operating systems, UWIN for K and Linux2.2.12 for F. The Sflo performance is slightly poorer on F than on K. To see if this difference is due to compilation environments, we reran the Sflo tests on platform K after recompiling with gcc version *egcs-2.91.66*, the same compiler on Linux2.2.12. The Sflo printing and scanning times on K are then 0.80s and 0.77s respectively. This shows that either gcc generates worse code than the native Microsoft-C compiler or its supporting libraries are not as optimized as the Microsoft-C ones, or both. Since the gcc-based timing results on K are also worse than that on F, it is likely that the support libraries for gcc on F are more optimized than on K.

- The printing performance of Stdio on platforms G, R and T is close to that of Sflo but its scanning performance is relatively much worse. This is particularly bad on T where Stdio scans data at a rate 4 times slower than Sflo. Since printing is more popular than scanning, perhaps these platforms recently improved their printing facilities though not the scanning ones.

- Platform D clearly has the worst performance in both printing and scanning. This is especially disappointing given the advertised processor speed. Since the timing results are similar between Sflo and Stdio, the poor performance must be due to the platform itself, i.e., the compiler or the support standard libraries.

The additional formatting features to define new patterns or redefine existing patterns do incur cost due to extra function calls. To see how much this cost might be, we wrote test programs to print a sequence of complex numbers whose real and imaginary parts are equal and range from 1 to *n*. For any given *n*, all programs produced identical output. Below are brief descriptions of the programs:

- **`sfio%c`**: This prints numbers using the method shown in Section 2.5.3.
- **`sfio`**: This prints numbers using the format string “`(%g,%g)`” in direct `sfprintf()` calls. For fair comparison with `sfio%c`, the program constructs `Obj_t` objects before using their parts in the printing calls.
- **`complex`**: This uses the `complex<double>` type in C++ and the output operator `>>` to print numbers to the standard output.

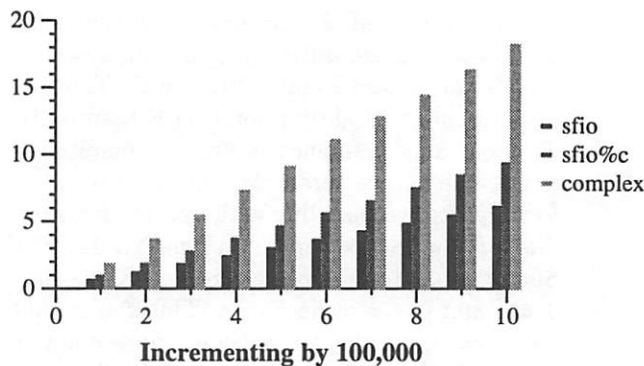


Figure 7: Times to print complex numbers

All programs were compiled on platform R using the compiler `g++` version 2.95. Figure 7 shows CPU+System times from test runs with `n` from 100,000 to 1,000,000 in increments of 100,000. Not surprisingly, the program `sfio` using direct `sfprint()` calls was fastest. The mapping of the new conversion pattern in `sfio%c` increased computation cost by about 50% relative to `sfio`. The program `complex` was slowest, about twice slower than `sfio%c` and three times slower than `sfio`. This shows that there is a significant performance cost to use the new extensions. However, this cost is not unreasonable in light of the cost incurred by a commonly used I/O facility in C++.

## 4 Conclusion

We discussed a number of extensions made to the printing and scanning functions in the Sflo library. These extensions enable safe and flexible manipulation of strings and scalar objects. Data formatting is fully generalized so that applications can provide their own interpretation of the conversion patterns and also define new ones. Examples were given to show how the new features enable applications that would be hard to build using Stdio.

A performance study was presented to show that, despite the additional formatting features, Sflo still performed as well or better than currently popular Stdio implementations when only standard formatting tasks are done. A separate experiment showed that the extended formatting features to define new patterns and/or redefine old ones could incur significant cost due to extra function calls. This cost should be balanced against the extra programming flexibility.

Although the Sflo's API is distinct from Stdio's, Sflo does provide source and binary compatibility

packages for programs written on top of Stdio. The extensions discussed here are orthogonal to the features defined in the ANSI-C Standard[1]. Therefore, they can be transparently used by Stdio applications that are compiled or linked with the compatibility packages provided by Sflo.

## Code availability

The source code for Sflo is freely available at:

<http://www.research.att.com/sw>

In addition, related commands and libraries are available at:

<http://www.research.att.com/sw/download>

## References

- [1] ANSI. *American National Standard for Information Systems - Programming Language - C*. American National Standards Institute, 1990.
- [2] Glenn S. Fowler, David G. Korn, J.J. Snyder, and Kiem-Phong Vo. Feature-Based Portability. In *Proc. of the Usenix VHLL Conference*, pages 197-207. USENIX, 1994.
- [3] Glenn S. Fowler, David G. Korn, and Kiem-Phong Vo. Sflo: A Buffered I/O Library. *Software—Practice and Experience*, Accepted for publication, 1999.
- [4] ISO/IEC. *ISO/IEC International Standard 9899:1999(E) Programming Language - C*. IEEE, 1999.
- [5] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice Hall, 1978.
- [6] David G. Korn. Porting UNIX to Windows NT. In *Proc. of the 1997 Usenix Conference*. USENIX, 1997.
- [7] David G. Korn and Kiem-Phong Vo. SFIO: Safe/Fast String/File IO. In *Proc. of the Summer '91 Usenix Conference*, pages 235-256. USENIX, 1991.
- [8] Posix - part 2: Shell and utilities, 1993.
- [9] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.
- [10] Kiem-Phong Vo. The discipline and method architecture for reusable libraries. *Software—Practice and Experience*, 30:107-128, 2000.



# Virtual Services

A New Abstraction for Server Consolidation \*

John Reumann†, Ashish Mehra‡, Kang G. Shin†, and Dilip Kandlur‡

†*Department of Electrical Engineering and Computer Science,  
The University of Michigan, Ann Arbor, MI 48109-2122*

‡*Server Networking and Network Services, IBM T.J. Watson Research Center,  
Yorktown Heights, NY 10598*

## Abstract

Modern server operating systems (OS's) do not address the issue of interference between competing applications. This deficiency is a major road-block for Internet and Application Service Providers who want to multiplex server resources among their business clients. To insulate applications from each other, we introduce *Virtual Services* (VSs). Besides providing per-service resource budgets, VSs drastically reduce cross-service interference in the presence of shared backend services, such as databases and name services.

VSs provide dynamic per-service resource partitioning and management in a manner completely transparent to applications. To accomplish this goal, we introduce a kernel-based work classification mechanism called *gates*. Gates track work that propagates from one service to another and are configured by the system administrator via simple rules. They automate the binding of processes and sockets to VSs, and ensure that any work done on behalf of a VS, even if it is done by shared services, is charged to the resource budget of the VS that requested it. Using our experimental Linux 2.0.36-based implementation we applied them effectively to co-hosted Web servers. Thus, nearly eliminating performance interference between the co-hosted sites.

## 1 Introduction

It is becoming increasingly common and desirable for companies to outsource applications and services to Internet or Application Service Providers (ASPs) to reduce hardware and administration costs. ASPs save cost

by sharing hardware, software licenses, and personnel among business clients. To minimize the number of system administrators required to run the ASP's site and to decrease the rate of system failure, ASPs invest in highly reliable and powerful servers. Since such server setups are not cheap, its resources should be highly utilized by sharing them among as many business clients as possible. In addition to the sharing of hardware resources, software resources may be shared across services. For instance, the DNS server will generally be shared across services and even business clients. While reducing the ASP's cost, aggressive sharing makes ensuring the Quality-of-Service (QoS) for the outsourced services difficult. Since ASPs must fulfill QoS contracts, a.k.a. service level agreements [22], they must tackle the performance interferences that result from the sharing of resources and services without deploying highly under-utilized and hence, cost-inefficient servers.

Recently, resource sharing has been addressed by the virtualization of resources in off-the-shelf OS's (e.g., *virtual Web hosting* and *virtual servers*) [2, 4, 6, 7, 8, 14, 21]. The essence of these concepts is that one physical server is split into several virtual hosts (VHs). Ideally, neither the client nor the server application is aware of the fact that it is executing on a VH and not on a real host. Initial implementations of this idea were content-based VHs. Here, a server would serve different content, depending on the IP address that was used to contact it, e.g., Apache's `VirtualHost` directive [14]. The performance interference that occurs between co-located VHs was not considered. To solve this problem, resource bindings for VHs were introduced [2, 4, 7, 20, 21]. With resource bindings, demand surges on one VH will no longer impact the performance of other co-hosted VHs. A service that is executed on one VH behaves as if it were executed on its own physical server. This still does not address the performance interference between services that may result from accessing the same backend service.

\*The work reported in this paper was supported in part by the IBM Graduate Fellowship Program and by the NSF under Grant No. EIA-9806280.

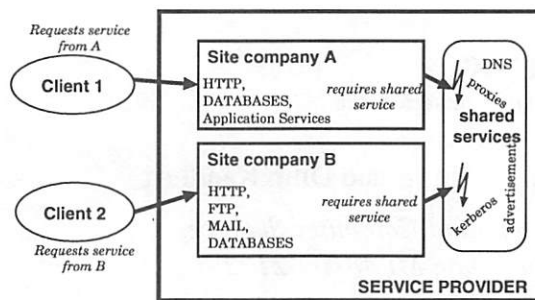


Figure 1: Service-sharing destroys insulation

Looking at the services that are hosted on a VH, we found that they vary between monolithic, which implement all service functionality themselves, and lightweight, which relay most of their work to other services. Therefore, the abstraction of a service does not directly coincide with the process boundaries imposed by the OS. Nevertheless, we define a *service* as the set of processes, sockets, and file descriptors that implement the service interface and share one address space. All other service activities (henceforth called just activities) that a service may trigger outside its own address space are referred to as *sub-services*. Since service providers often host similar services for different business clients, sub-services often shared among them.

When services are shared among different business clients, VH-based insulation approaches fail (see Figure 1). Shared services like DNS, proxy cache services, time services, distributed file systems, and shared databases, are quite common. With shared services, an obvious question arises as to which VH should host these shared services. Since these services work on behalf of many other services, their resource bindings should be dynamic to reflect the *works-for* relation. The problem could be fixed by replicating shared services on each VH. However, in this case the consistency of individual shared services becomes a major concern, as does the inefficiency that results when hosting two identical services to maximize performance insulation.

To eliminate the performance interference caused by shared services, we introduce the notion of a *Virtual Service* (VS). VSs are an OS abstraction that provides per-service resource partitioning and management by dynamically binding service activities in a manner that is completely transparent to applications. Once an activity is classified as belonging to some VS, this VS association is maintained, regardless of the process context in which the activity continues. This means that the resource bindings for shared services are delayed until it is known who they work for. This automatic and delayed resource binding makes insulation between services pos-

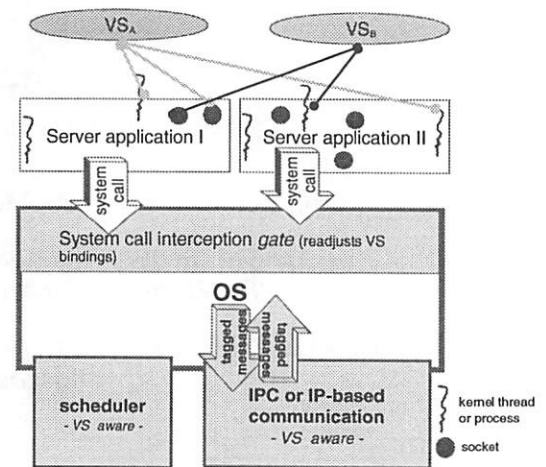


Figure 2: Virtual Service Architecture

sible, in spite of shared sub-services.

In the VS architecture, the dynamic binding of activities to VSs is inferred by intercepting system calls within the OS using *classification gates* and analyzing the information that is passed to the call (see Figure 2). Gates track work that propagates from one service to another and are configured by the system administrator via simple rules. They automate the binding of resources (e.g., newly-created processes and sockets) to VSs, and keep track of any work done on behalf of a VS. Administrators can specify which system calls should be intercepted. Furthermore, they set up the rules that specify how the association between resources (e.g., processes, sockets, etc.) and VSs is affected by the intercepted calls. For example, a rule like: "If process  $P_1$  accepts a service request from  $VS_x$ , the resulting  $P_1$  activity should be charged to  $VS_x$ " can be configured easily. Our design also includes a VS-aware scheduler and packet dispatcher that enforce per-VS CPU and network shares.

The combination of VSs and gates permits resource management for dynamic communicating services without requiring any changes to the hosted services. While being transparent to the application, the system administrator must know the mechanisms by which work is relayed among VSs. Fortunately, this information can be inferred without application modification by system-call-tracing utilities or packet sniffers.

Other approaches that permit changing the resource bindings [2, 4] do not consider the problem of shared sub-services that do not readjust resource bindings on their own. Since the applications that the ASP hosts are usually developed for standard OS's, they are not resource-binding-aware. As Figure 2 shows, our transparent VS architecture brings resource management to such applications.

The following points summarize the key features of our

VS architecture:

- Dynamic resource bindings for shared services
- Application transparent resource management
- Applications may use several OS mechanisms to relay work to each other
- Minimal interference between VSs
- Modular implementation permits trade-off between the quality of insulation and the overhead incurred by the VS abstraction

The idea of a VS can be directly applied to distributed server farms within one administrative domain and shared VS name space. Even though collaborating services may be scattered over several machines, they still relay service requests to one another via system calls which the classification gates can intercept.

We summarize related work in Section 2. Section 3 details the design of VS and the dynamic management of VS associations. Section 4 describes our Linux-based implementation. Section 5 presents experimental results and quantitatively shows that the VS abstraction solves the problems faced by ASPs that co-host services. Section 6 summarizes our findings and future research directions.

## 2 Related Work

There are two approaches to server management: *resource-oriented* and *service-oriented*. Resource-oriented approaches like Resource Containers [2], Eclipse [3, 4], Capacity Reserves [15], and Hierarchical Scheduler [9] provide necessary low-level support for the partitioning of resources. Furthermore, they support relatively static bindings of resource consumers to these partitions. VS (proposed in this paper) and Workload Manager [1] are service-oriented. They charge services for their resource usage instead of creating static resource partition bindings for entities like processes, users, or sockets. Table 1 characterizes the properties of related approaches. This figure also highlights the novel features of VS.

Resource Containers (RC) is the most recent representative of resource-oriented server management. Any OS or application activity executes in the context of an RC. The RC may contain basic CPU and network shares and various count limits on the number of resource consumers that can be bound to it. To control application performance, processes must explicitly bind to an RC. Subsequent activities are charged to the associated RC, and resource limits specified therein are enforced. Unlike

VS, the RC abstraction does not automate the binding of resource consumers to RC's.

The Solaris Resource Manager [20] is based on a resource reservation concept (called *lnodes*) which is equivalent to RC. In addition to the resource-reservation abstraction, *lnodes* are tagged with Unix user-group affiliations so that the resource context can be inferred from the user-group setting of current application activity. This mechanism reduces the need for manual resource bindings. The idea is to give each user-ID its own machine. Unfortunately, this concept fails if shared services do not change their user-ID when they work on behalf of different users. For example, the system's DNS server does not change its user-ID to that of the process requesting address resolution.

In the context of Eclipse's hierarchical reservation domains [4] Bruno *et al.* discuss in a recent paper [3] how Eclipse tackles the problem of sharing specific OS entities such as sockets among concurrent applications. Interference can be reduced by tagging each request that utilizes a shared resource with the appropriate reservation domain, thus delaying the resource binding of the shared OS abstraction. Request tagging is also used by VS. Unlike VS, Eclipse does not infer the tag for a request in the absence of application support and does not exploit these for the scheduling of an application that picks up a tagged request. Precursors of this work are the Hierarchical Scheduler (HS) [9] with configurable CPU scheduling policies and the Nemesis OS [10]. Nemesis provides comprehensive inter-application isolation for memory and file system. Both HS and Nemesis require applications to manage their own resource bindings.

Workload Manager's (WLM's) [1] notion of a *service class* is similar to the notion of a VS. Since WLM manages requests separately according to their *service class*, service sharing does not necessarily cause interference. Nevertheless, classifying requests into service classes is the hard part. For this purpose, IBM modified OS/390's services to classify all requests into service classes. This approach does not work for ASPs since they cannot modify hosted applications. Therefore, VSs provide a transparent work classification mechanism.

Scout [18] takes a somewhat different route since it is primarily designed to be used in embedded multimedia server designs. Scout's path abstraction tracks the flow of work across different OS layers. Resources are reserved on a per-path basis. Since paths are compiled into the kernel, resource consumption scenarios cannot change dynamically. For every new resource consumption scenario (i.e., new applications) the Scout kernel must be recompiled. In contrast, VSs can be configured dynamically to handle new scenarios of resource con-



	Dynamic System Domains	Eclipse BSD	HS	RC	Resource Manager	Scout	VS	WLM
OS	Solaris	Eclipse	Solaris	Digital Unix	Solaris	Scout	Linux	OS/390
Focus	VH on multi- processor	VH	Multimedia end-host	abstract VH	per-user VH	Multimedia end-host	per- service resources	workload mgmt
Single server	Y	Y	Y	Y	Y	Y	Y	Y
Server farm	Multi- processor	N	N	N (pot. Y)	Multi- processor	N	Y	Y
CPU control	Y #CPU's/VH	Y	Y	Y	Y	program- mable	Y	priorities
Net control	Y (coarse)	Y	N	indirect	N	program- mable	Y	N
Disk control	Y (coarse)	Y	N	N	N	program- mable	N	Y
Memory control	Y (coarse)	N	N	N	Y	program- mable	Not yet	Y
Inference of resource principal	VH	N	N	some TCP- based	Unix user/group	N	rule-based Y	app-based
Recognize work delegation	N	N	N	N	N	hardwired into OS	intercept syscall	N
Application changes	Y	N	Y	Y	Y	OS part of application	N	Y

Table 1: Resource- and service-oriented server management solutions

sumption and service interaction.

Sun's Dynamic Enterprise 1000 [21], Solaris Resource Manager [20], and Ensim's recent VH product ServerX-change [7] are noteworthy commercial VH implementations resembling Eclipse. Other popular commercial solutions, such as Cisco's LocalDirector [6], HydraWeb [11], and F5's BigIP [8] are geared towards increasing the capacity available to ASPs through load-sharing in server clusters. These solutions also provide some coarse insulation between co-hosted services by shaping request flows. This mechanism applies only to well-known services (e.g. Web servers) since requests must be parsed by a load-managing frontend. Hence, insulation fails when ASPs co-host proprietary services and when the workload created by individual requests differs significantly among co-hosted sites.

### 3 The Virtual Service Abstraction

The VS abstraction treats services that utilize sub-services as if they were a single application executing on its own dedicated server. To create this illusion, a VS is associated with a basic resource context (Figure 3).

The resource context summarizes the resource limits and statistics for activities that execute on behalf of the VS (Section 3.2). Figure 3 also shows typical VS members. Section 3.3 discusses how to track this dynamic member set if applications do not manage VS-membership on their own. In Section 3.4 we discuss the seamless integration of the tracking of VS-membership and resource limit enforcement into a *gate* abstraction. A gate filters entry and exit of system calls that are relevant with respect to VS-membership changes. Section 3.5 explains the gate's response to resource limit violations.

#### 3.1 System Model

Our approach uses tags OS entities, such as processes, sockets, IPC shared memory segments, etc., with VS information. All modern OS's have these entities and already tag them with other information. We also have to assume that all requests for service are received via explicit system calls, such as communication sockets, IPC shared memory segments, message queues or pipes. This restriction is due to the fact that automatic tracking of changes in VS-membership depends on being able to intercept the interaction between cooperating services.



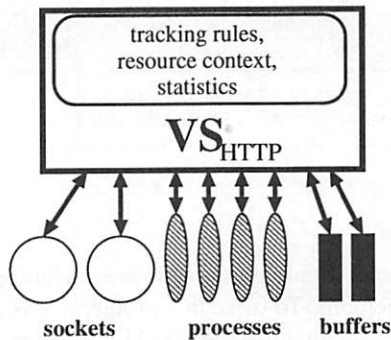


Figure 3: A VS and its members

This means that VSs cannot manage services that hide their relaying of work. For example, if two services use some VS-unaware server as a message relay, it is impossible to infer their cooperation. User-level thread libraries are another form of hidden communication since the application's switching between different requests is hidden from the OS. Here, the thread library must rebind the process to the correct VS every time a thread-switch occurs.

### 3.2 Setting up a Virtual Service

Each VS is uniquely identified by its descriptor (Figure 4). To allow the system administrator to manage VSs, each service has an integer virtual service identifier (VSID). The VSID is guaranteed to be unique on each machine. For VSs that use resources on multiple machines, we leave it up to user-space software to guarantee the uniqueness of identifiers. When setting up the distributed service, external administration software can force a specific VSID onto the newly-created VS. Such global VSID's are taken from their own number range and will never conflict with local VSID's.

Like RC's [2] we advocate hierarchically-nested resource contexts. Hierarchy is necessary because an ASP's clients should be able to decide themselves whether they want all of their services to share resources or whether they want to insulate them from each other.

The parent field of the VS structure points to the parent VS. Oftentimes parent VSs will be used to implement abstract VSs, i.e., placeholder services to which all services of an ASP's business client belong. The highest-level service is the `root_service` with VSID 0, which accounts for all unclassified work.

Hierarchy is again reflected in VS attributes such as resource usage statistics and resource limits. By default, newly-created VSs share the attributes, i.e., resource control settings and statistics (CPU time used, packets

sent, etc.) of their parents. This means that the fields in the child refer to its parent's pendants (Figure 4). To manage the child service directly, attributes of interest need to be detached from the parent. For example, to control the number of processes for a sub-service, one detaches the sub-service's process count limit via the `servctl (DETACH_PROCESS_LIMIT)` call.

To instantiate a cluster-wide VS, the administration software must create VS descriptors with one global VSID on all cluster nodes. On each of those nodes, local resources may be reserved using the VS descriptor's resource context. Before reserving VS resources, the administration software will monitor the VS's resource consumption via the statistical VS attributes. Once enough statistics are available, resource reservations will be calculated to stabilize VS performance. This calculation is difficult and requires a full-fledged monitoring-feedback algorithm, which is outside the scope of this paper. We experienced that cluster-wide VS management using a unified VSID name space simplifies the implementation of such an algorithm.

Most of the VS state discussed so far could potentially be realized using RC's [2] or Reservation Domains [4]. However, they do not provide configurable classification rules. Classification rules indicate how VS-membership is to be updated when certain system calls are invoked by specific VSs. For instance, if a process member of the VS in Figure 4 calls `fork`, the OS knows exactly that this is a way of relaying work and that the created process should inherit the parent's VS affiliation.

### 3.3 Tracking Service Membership

There are two ways of tracking the members of a service: either they are announced or the OS infers who they are. In our VS framework, membership is mostly tracked by the OS without requiring continuous application or administrator intervention. Nevertheless, especially at service startup time it can be efficient to create some associations between VSs and other OS entities explicitly. For example, if one knows that one specific kind of service request (identified by its own VS) always enters the system through one specific process or socket, a manual binding of these processes or sockets to the VS should be used. This avoids having the OS infer repeatedly that these entities and the VS belong together.

As a VS begins to respond to requests, new sockets, processes, and IPC resources may be created. Each of them must be associated with a VS because they incur system load and are used to relay work. Usually these new members are not added explicitly since the administrator does not even know of their existence and the applica-

VSID	resource limits	resource statistics	classification rules	rule priority	parent
global 123	CPU rate — comm rate ...	actual CPU use actual net use ...	fork-ed children map to VSID 123(error EAGAIN) connect-ed service maps to VSID 123(error none)	1	global 120

may refer to parent attribute

Figure 4: The VS descriptor

tion does not cooperate with the VS abstraction. Therefore, membership for these new entities is implicitly determined by the classification rules in the creator's VS descriptor.

Not only do new entities need to be associated with a VS, but VS-membership may also change over time. For instance, if some process is observed to be operating on a particular data set that is characteristic of some separately-managed VS, the process is added to that VS and removed from its current VS.

Classification, i.e., associating an OS entity with a VS, takes place when the OS can infer something about the application, i.e., at system call time. We can limit VS-membership inference to those times because we assumed in Section 3.1 that VSs interact with each other over a limited set of OS mechanisms. This means that the *works-for* relation cannot change unless a system call is invoked. Therefore, there is no need to update VS-membership at any other time.

The **classification rules** that the OS examines at system call interception consist of a conditional clause, which defines when the classification rule is applicable, and a classification directive. This is formalized as:

$$(\text{syscall}, S_1, \dots, S_m, P_1, \dots, P_n) \rightarrow (S'_1, \dots, S'_m)$$

Here  $S_i$  represents the VS of the  $i$ -th affected entity. For example, the only affected entity in the `exec` call would be the calling process. The calling process's VS is always identified by  $S_1$ .  $P_j$  represents the  $j$ -th intercepted property, for example, the program name passed to `exec` or the incoming IP address of an `accepted` connection. Properties are not OS entities. A classification rule also specifies  $S'_i$ : the resulting VS classification of the  $i$ -th affected entity. This classification is applied only if the conditional (left-hand side of the rule) matches the intercepted system call.  $S_i$  and  $P_j$  may be wildcards. Our prototype implementation requires  $S_1$  to be specific. The system call is always specific, since the dimensionality of the condition tuple depends on it. One way of managing the rules would be to store them in a system-wide table, which would ease the integration of VSs into RC's or Eclipse. Our implementation reduces lookup times by storing rules solely inside the VS descriptor that matches  $S_1$ .

**Conflicting rules:** Rule matching can lead to ambiguity.

Multiple condition tuples may match the current system call interception. To solve this problem, VSs are prioritized. The rule that matches the highest-priority VS explicitly is used to determine the resulting classification. Should there be a tie between several rules, the most specific rule is applied. If this does not resolve ambiguity, the result is unspecified.

In the remainder of this subsection, we will discuss the different types of classification triggers. Table 2 casts common UNIX system calls into these categories.

**Creation:** If an entity,  $A$ , creates another OS entity,  $B$ ,  $B$ 's future VS affiliation depends solely on  $A$ 's VS affiliation. Examples are the creation of sockets, IPC shared memory segments, message queues, pipes, and the like. The canonical default rule is for the created entity to inherit its creator's VS affiliation.

**Communication:** Communication is used to relay work within and beyond machine boundaries. Therefore, intercepting intra-VS communication is essential to VS maintenance in server farms. If it is possible to determine the VS affiliation of each request that is picked up by a service, the resulting activity can be charged to the correct VS. This does not depend on whether the request originated locally or remotely.

Communication affects at least three entities: *sender*, *receiver*, and the *message* itself. To make inter-process communication more efficient, most OS's implement asynchronous communication as opposed to the *rendezvous* concept. This adds sockets, pipes, and the like to the set of affected entities, each of which may be reclassified upon system call interception.

Due to the temporal separation between sending and receiving a message, pausing to reclassify the affected entities is difficult. Therefore, communication-based VS tracking is done in two stages. First, when the message leaves the sender it is tagged with a VS affiliation, much like what is done in the case of creation-type calls. This can be skipped if the communication is a one-to-one connection. In this case, the connection itself is labeled at setup time with a VS affiliation that implicitly applies to each message that passes through it. The second stage is message consumption. At this time, the receiver's VS affiliation may change based on its previous

Category	System call	Proposed classification mechanism	Proposed VS error	Used by
creation	fork	Classify new process based on forker's VS	EAGAIN, block	Multi-threaded services
	open	Tag created file descriptors with a service affiliation based on creator's VS	block	User services, FCGI
	socket			All network services
	pipe	Tag new shared memory segment based on creator's VS	EINVAL, block	User services, FCGI
	shmctl			
communication	accept	Classify caller based on its VS, the VS of the incoming connection and incoming source address	EWOULDBLOCK, block	User services, FCGI
	connect	Classify caller and connecting socket based on the destination IP + port	EINPROGRESS, block	Frontend services, HTTPD with distributed FCGI
	send	Classify caller and sending socket based on destination IP + port	EWOULDBLOCK, block	Frontend
	sendmsg			
	sendto	Classify caller and receiving socket based on incoming packet's VS and IP + port	EWOULDBLOCK, block	NFS, DNS, RPC, Multimedia
	recvmsg			
	recvfrom	Classify caller based on shared memory's VS	EACCESS, block	Apache (thread synch)
	recv			
	shmat	Tag message based on caller's VS	block	Proprietary services
	msgsnd	Classify caller based on the incoming message's VS	ENOMSG, block	
synchronization	semop	Classify caller based on the semaphore's VS	block	Proprietary services
transformation	exec	Classify caller based on the executed program's name	block	HTTP-CGI, Inetd, rexec, rsh
	listen	Classify caller based on its and socket's VS		TCP + Unix Domain socket-based services (Fast CGI)
	bind	Classify caller and socket based on the IP + port pair		Standard services
communication, synchronization, transformation	write	Tag the message based on caller's VS or inherit file descriptor VS	EAGAIN, block	All services
	read	Classify caller based on read message's VS or inherit file descriptor VS		

Table 2: System calls that affect VS-membership

VS affiliation and the received message's VS affiliation.

**Synchronization:** The set of affected entities in synchronization includes the executing process(es) and all process(es) in the wait queue for the synchronization primitive. Activities that are performed under the protection of a synchronization primitive may be associated with its VS.

Synchronization can also be used to infer collaboration among a set of processes. Previously-unclassified processes may inherit the VS affiliation of the synchronization primitive. This is an effective tool since many multi-threaded server applications expose their process sets when they synchronize for thread control purposes.

The process(es) that execute under the protection of the synchronization primitive must not stall processes in the wait queue because otherwise, priority inversion [13] will result. This is also a problem when a single-threaded sub-service is shared among several VSs. This will be discussed further in Section 4 (accept).

**Transformation:** Whenever the kernel intercepts a characteristic argument to a system call, it is possible to classify the caller and other affected entities more accurately. For instance, the program name in the `exec`-call

allows a more accurate VS classification of the active process if the program is typical of a specific VS. Other frequently-used system calls that may alter VS classifications without relaying work are `setgid`, `setpgrp`, and `setuid`.

### 3.4 Virtual Service Gates

Whenever a VS receives a new member, resource limits could potentially be violated. This means that classification and resource limit enforcement are inseparable. Therefore, we introduce *gates*, a combination of system call filtering and VS classification. Each system call that is used to track VS-membership is controlled by a gate.

If the gate's filtering code indicates a resource limit violation as a result of the new classification, the system call will either fail with an administrator specified `errno` code, block, or execute in best-effort mode. Otherwise, VS-membership is updated as specified in the classification rules. Figure 5 depicts the basic anatomy of a gate:

1. The *prefilter* checks whether the caller is (a) classified and (b) whether its VS affiliation permits the execution of the gated call.



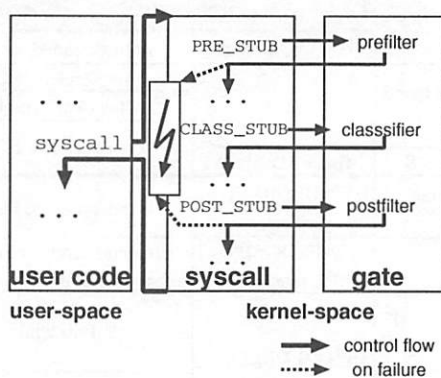


Figure 5: Gated system calls

2. The *classifier* applies a matching classification rule. To execute the classifier for creation-type calls it executes after the new resource has been created.
3. Finally, the *postfilter* checks whether the resulting classification violates any VS resource limits. The resource limits we considered are: count limits on the number of processes and sockets. Other resource limits, such as CPU and network bandwidth are enforced silently by the packet and CPU schedulers and need be checked by the gate mechanism. If a resource limit is violated, the system call fails or retries as is described in the next section.

### 3.5 Failing System Calls

Gates may detect resource limit violations. For example, during the execution of `fork` it may become apparent that successful call completion would result in a violation the VS's process count limit. The appropriate remedy is application-dependent. One may decide to:

1. Wait until VS resources become available.
2. Return an error to the caller indicating resource exhaustion.
3. Not apply the classification that led to the resource limit violation and silently reclassify the caller as best-effort. If the best-effort VS has exhausted its resource share, there is no other option but to fail the system call.

In the first case, the OS will add the caller to a FIFO wait queue for the requested resource. For example, in the case of `fork` this means that the forker will sleep until the VS's process count drops below the process count limit. The resulting delay may not be acceptable to the calling application.

Applications that cannot be delayed, should receive an error upon resource exhaustion. Unfortunately, existing applications may not be able to handle arbitrary errors. Therefore, we leave it up to the administrator to configure the error that will be raised if a VS-level resource limit violation is observed at a particular gate. In this way, only errors that the application is able to handle will be raised. For example, the administrator may choose to raise the `EAGAIN` error for some VS that exceeds its process count limit upon `fork`. This behavior is specified at the time of gate configuration [e.g., `servctl(SETFORK_POLICY, VSIDx, ..., EAGAIN, ...)`]. Most server applications are capable of handling errors that result from resource exhaustion gracefully. They simply record the error in a server log-file to support system tuning. If neither blocking nor returning an error is acceptable to the hosted service, the execution should continue in best-effort mode.

## 4 Implementation

We added VSs as loadable modules to the 2.0.36 version of the Linux kernel; located at <http://www.eecs.umich.edu/~reumann/vs.html>. Figure 6 shows dependencies among the VS modules. To implement the gates, we added only a few lines of call-back code to the intercepted system calls to trigger VS classification. The VS structure itself (Figure 8) contains the previously-described membership information, statistics, and resource limits. The VS structure, VS hierarchy management and most of the gates are portable since they hardly depend on Linux internals. The placement of the call-back code in the original system calls is Linux-specific.

We implemented VS-level *fair-shares* [12, 15] for CPU and network to provide strict VS-level resource guarantees. VSs that are neither directly nor indirectly (via a parent VS) associated with a share are scheduled on a best-effort basis. Best-effort VSs use all unreserved resource slots. Any excess capacity is shared between VSs that own resource shares in a round-robin fashion (see also firm Capacity Reserves [16]). The implementation of VS resource shares is not portable across platforms. Nevertheless, numerous implementations of capacity reserves and fair-shares exist. Therefore, requiring VS-level fair-shares does not limit the applicability of our approach.

VS statistics are cumulative aggregates of the VS's members' statistics. The attributes include a wide range of statistics that Linux keeps for processes and sockets, such as page faults and virtual time elapsed.

To set up the VS hierarchy and adjust CPU limits,



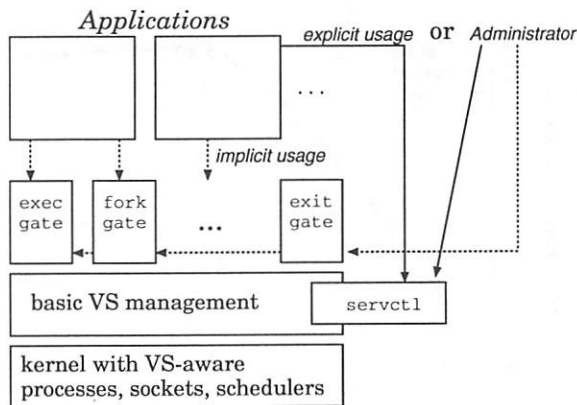


Figure 6: VS module dependencies

VS membership, policies, attribute inheritance, resource limits, and query VS attributes, the OS offers a new system call (`servctl`). It takes a command, the size of the argument, and an argument structure as parameters.

Gates are implemented as loadable modules. We currently support `fork`, `exit`, `exec`, `open`, `accept`, and `socket` gates. Upon insertion of a gate module, the call-back stubs that are placed in their corresponding system calls are activated so that the gate's *prefilter*, *postfilter*, and *classifier* are executed each time the control flow of a server application passes through the intercepted system calls. Each gate also registers its own `servctl`-handler to enable gate configuration.

The advantage of our modular gate design is that one only needs to add those gates to the kernel that are absolutely necessary to classify VS membership and insulate services. This is very important because the insertion of each gate into a running kernel increases system overhead (see Section 5 for more detail). The remainder of this section describes our implemented gates.

**Fork:** Upon interception of `fork` the created process is classified as a new member of some VS. To determine the resulting VS affiliation, we check the `fork_policy` object of the creator's VS. The `map_to` attribute of the `fork_policy` specifies the affiliation of the created process. If the VS specified by `map_to` has reached its process count limit (set via the `servctl` call), the failure behavior that was configured for that VS is invoked (Section 3.5). Figure 7 shows a high-level control flow graph for this gated system call.

**Exit:** If a process exits — including ungraceful `SIGSEGV` and other uncaught signal exits — it must be removed from the VS with which it is associated. This gate is not configurable.

**Exec:** Upon calling one of the `exec`-family system calls, the caller can be reclassified based on the name of the program that was invoked. The gate code checks

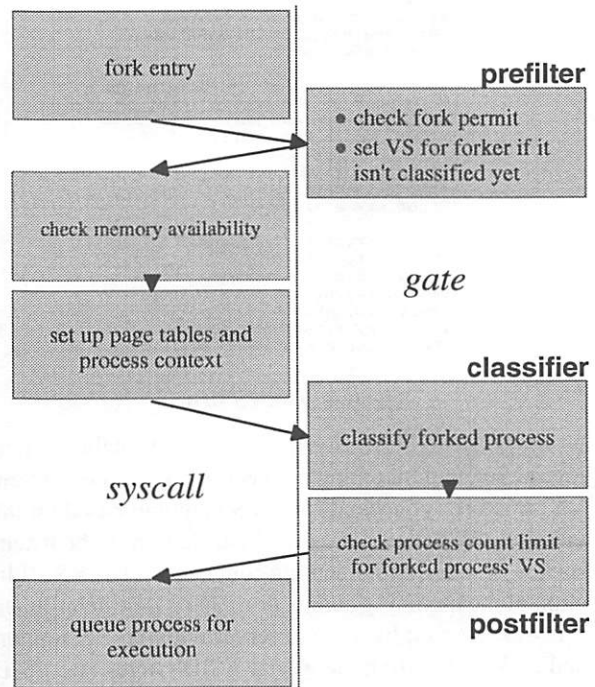


Figure 7: Control-flow of the fork gate

the name of the program against a hashed mapping table, i.e., the `exec_policy` field in Figure 8.

**Open:** The `open` gate acts like the `exec` gate. The only difference is that the file descriptor may be tagged with a VS affiliation at the same time. Moreover, the `open` gate uses a prefix-tree to match the file names. Thus, whole directories — identified by a shared prefix — can map to one VS. This is important because large numbers of data files residing in one directory subtree may yield identical VS classifications.

**Socket:** The `socket` gate resembles the `fork` gate. The `socket_policy` of a VS specifies the future VS affiliation of the created socket. Once messages are relayed via such a classified socket, they are tagged with the VS affiliation of the socket in their IP Type-of-Service field (TOS), thus allowing VS information to propagate over the network. Since the TOS field may be used by Diff-Serv to provide differential QoS in a WAN, this field can only be used inside server clusters. If the TOS field cannot be used or one needs more than 256 VSID's (the TOS is eight bits wide), one may introduce a new IP-option [17] to hold the VSID. This option should be set in every fragment of the IP datagram to facilitate VS-aware routing.

**Close:** Closed file descriptors' and sockets' VS affiliation must be removed.

**Accept:** The `accept` gate is quite complex. It first de-

```

struct service_struct {
    int sid;
    struct service_struct *parent;
    char name[MAX_SERVICE_NAME_LEN];
    int precedence;

    // int_or_ptr is either a value or a
    // pointer to the parent's int_or_ptr
    int_or_ptr process_count;
    int_or_ptr socket_count;
    int_or_ptr byte_count;
    int_or_ptr vtime;
    int_or_ptr majflt;
    int_or_ptr minflt;

    member_struct *processes;
    member_struct *sockets;
    member_struct *services;
    fork_policy_struct fork_policy;
    exec_policy_struct exec_policy; ... more ...
    cpu_policy_struct cpu_policy;
    comm_policy_struct comm_policy;
};

```

Figure 8: The VS struct

termines the highest-priority VS among the caller, listening socket, and incoming connection. Then the winning VS structure is checked for a VS mapping based on the incoming IP address and the VS affiliation of the listen-socket, process, and incoming connection. The VS affiliation of the incoming connection can only be determined if it was initiated by another server with our VS support and its VSID is from the global VSID range. The VSID is stored in the incoming SYN packet's IP TOS bits. For local accepts, the VS of the incoming connection is the connecting socket's VS affiliation. Both socket and receiver may be reclassified.

The difficulty with `accept` is that it should not block if the first pending connection on the listen queue leads to a violation of resource limits. There may be a connection that can be accepted without violating any VS resource limits. Therefore, our implementation scans the listen queue for the incoming connection whose VS has utilized its resource reservation the least.

**Concurrent Gate Versions:** A powerful feature of our implementation is to allow multiple versions of a gate to be loaded at the same time. Hence, VSs may specify which gate version they want to use when their process members invoke the corresponding gated system call. This way it is possible to eliminate unnecessary checks for specific VSs. For example, if `fork`-ed off processes should always inherit their parents' VS affiliation, it is unnecessary to check for a  $(\text{fork}, \text{VSID}_x) \rightarrow \text{VSID}_x$  mapping as is required for general VS classification. One can implement one `fork` gate version that always applies the parent's VS affiliation to the `forked` child. Another example is the `accept` gate, which is quite complex in its general form. In a server-farm setup it is likely that incoming service requests are already classified by the frontends and that the applications that process requests in the backends only need to inherit these classifications. This reduces the complexity of the backends' `accept` gates. We used such an op-

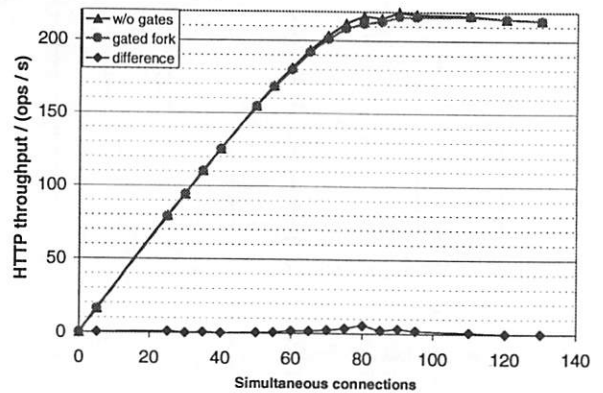


Figure 9: VS effect on HTTP throughput

timized version of the `accept` gate in our experiments. In our experiments incoming requests were classified as they were picked up by the HTTP server. Whenever the HTTP server relayed work to a shared backend Fast-CGI (FCGI) service, the backend FCGI inherited the classification of the requesting HTTP server process.

## 5 Evaluation

We evaluated the performance of the VS architecture on a small Web server running on a Dell 450 MHz Intel Pentium II PC with 448 MB RAM and one UDMA HDD. The clients, three 300 MHz Pentium II machines with 128 MB RAM each, were connected through 100Mbps Ethernet. We measured the performance of Apache 1.3.6 (HTTP 1.1) on this platform running on the Linux 2.0.36 kernel. The workload was generated by the commercial SpecWeb99 benchmark [19]. SpecWeb99 attempts to model a realistic workload including 30% dynamic requests. The size of the file sets grows linearly with the number of simultaneous connections offered to the Web site. Therefore, it generally does not completely fit into the server's file cache. Dynamic requests and the use of Apache explain the low HTTP throughput of the server (ca. 220 ops/s). Since the VS abstraction is an application-transparent mechanism, neither applications nor libraries had to be modified. The management of the VS hierarchy and gate configuration was done from the command line using utilities that feed their arguments into the appropriate `servctl` call.

### 5.1 Baseline Performance

Basic performance measurements show that the dynamic VS classification layer degrades overall system performance only minimally. If we intercept a complex sys-

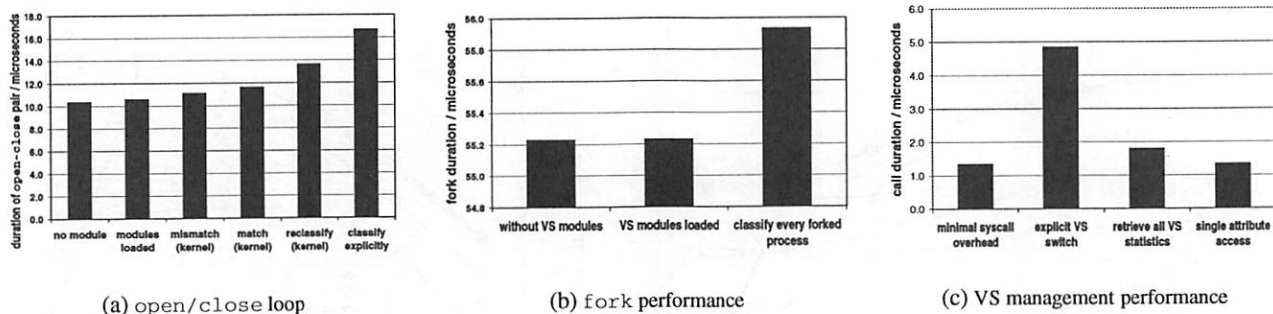


Figure 10: Performance of intercepted and new system calls

tem call like `fork`, the overhead of classifying the new process is small — only 1.3% — (see Figure 10(b), `classify`). Nevertheless, the raw performance of intercepted system calls can decrease significantly if the intercepted call is very simple like `open`. We observed 30% cost increase for the `open/close` pair if the VS affiliation changed with every execution of the loop (`reclassify`) in Figure 10(a). Just finding a classification rule (`match`) or not finding one (`mismatch`) is much cheaper. The high relative overhead for simple calls results from the almost constant classification overhead. An important point shown in Figure 10(a) is that binding processes to VSs from user-space (explicit classification) performs much worse than kernel-based classification because of the system call overhead. Explicit classification requires the executing process to classify itself and the resources that it uses and creates by calling the `servctl` system call. In our measurements we compared the performance of a sample program using (implicit) classification rules against a modified version of the program that explicitly updated affected VS bindings. The performance numbers strongly support the use of kernel-based (implicit) classification. For the sake of completeness, Figure 10(c) summarizes the cost of querying VS attributes and administering the VS hierarchy.

To estimate the overall performance impact of the kernel modifications including scheduler changes, resource limit enforcement, and the cost of system call interception, we measured how the performance of the Apache HTTP server [14] is affected by the OS changes. Figure 9 shows that the VS abstraction affects the system's HTTP performance only up to 2.5%, depending on the number of simultaneous client connections. The bi-modal shape of the performance loss graph in Figure 9 can be explained as follows. Apache keeps some spare processes alive to serve incoming connections faster. Once the number of simultaneous connections offered to Apache increases to an extent that there are not always enough of these spares, Apache begins to fork more connection-handling processes on-demand, which explains the increase in overhead up to 80 simultaneous

connections. Beyond this point, the file system cache hit ratio goes down so that the low performance of the file system begins to dominate overall system performance, thus decreasing the relative impact of our OS changes.

## 5.2 Implementing VHs using VSs

Another series of experiments on Apache shows that the VS abstraction may be used to insulate VHs. Unlike other applications, Apache itself provides some basic resource controls (process count limits) to insulate VHs. We studied the insulation properties that Apache can provide in comparison with those of VSs. The goal was to divide the previously-measured server into two VHs of equal capacity.

In the experiments, two copies of Apache were executed on the same host, each listening on its own IP address using IP aliasing for the Ethernet interface. Running two copies of Apache, each instance can be controlled by adjusting the `MaxClients` directive, which limits the number of concurrent sessions for each site. This is an effective means of performance insulation if the average work per HTTP operation is known for each site. Since both sites have their own copy of similar content, we could achieve a good division of resources by setting the `MaxClient` directive for the Apache servers to the same value.

To test VS-based insulation, the Apache servers were launched as if they were run on their own physical hosts (using very large process limits). We created two VSs, `www1` and `www2`, for which we specified the `fork` classification rules:

```
(fork, www[1|2]) → (www[1|2])
```

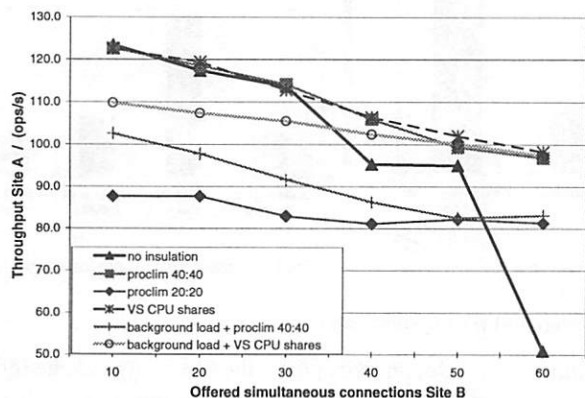
Each site's initial `httpd` process was explicitly added to its corresponding VS via a simple command line utility:

```
$> svcaddprocess <VSID> <PID>
```

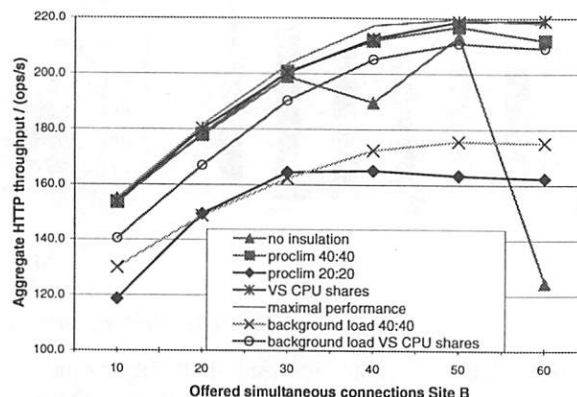
Each site was given a 50% CPU share.

In the measurements that are discussed here, Site A was offered a constant load of 40 simultaneous connections





(a) Site A performance in a two-site setup



(b) Aggregated performance in a two-site setup

Figure 11: Performance loss when hosting two sites of equal capacity on one server

while Site B was offered between 10 and 60 simultaneous connections. We chose these parameters because the server saturates — diminishing HTTP throughput gains — when offered 80 simultaneous connections.

Without insulation between the sites, A's performance degrades significantly once the server is offered a total of 70 simultaneous connections ( $A=40$ ,  $B=30$ ) [see Figure 11(a)]. From this point on, B begins to steal resources from A, thus contaminating the file cache to A's disadvantage. The lack of insulation can be fixed in Apache itself by restricting the maximal number of concurrent processes. This comes at the expense of some loss of aggregated performance under peak load [Figure 11(b)]. This loss is due to the fact that incoming requests must be rejected when the process limit is reached. This queuing phenomenon — for M/M/m/c systems described by the Erlang-loss formulas [23] — is especially evident when looking at the smaller process count limit (20:20). VS CPU shares eliminate this problem.

Apache's process limits also fail when background activities compete for CPU time, e.g., monitoring. To simulate the effects of background load, ten background load generators were invoked. As expected, aggregated performance and A's performance drop significantly if Apache's process limits are used for site insulation. In contrast, the VS abstraction keeps A's performance stable since only non-dedicated resource slots (beyond A's and B's resource limits) are used to process background load. Therefore, VS-based insulation performs better than Apache's own support for VHs.

One may argue that a modified, CPU-share-aware Apache could achieve the same quality of insulation. VSs obviate the need for modifying applications to get a

better handle on performance management.

Since this experiment did not involve access to any shared services and work is relayed only from a parent process to its child, Eclipse or RC's could probably be tuned to perform just as well as VSs. Beyond establishing the competitiveness of the VS approach, the next set of experiments focuses on its main contribution.

### 5.3 Insulation Despite Service Sharing

Instead of letting the sites A and B execute CGI scripts to serve the advertisement banners (part of the benchmark), a shared, single-threaded Fast-CGI server (FCGI) was used. Queuing theory suggests that the impact of this shared FCGI will be the worst when (a) it exhibits highly variant execution times and (b) a high percentage of requests are forwarded to it. Therefore, we modified the FCGI to execute a busy wait cycle randomly chosen between 0 and 10 ms (uniformly distributed), before serving incoming requests for advertisement. Furthermore, the percentage of advertisement banners requests was raised from 13% to 30% on each site. Other dynamic requests were eliminated from the benchmark's workload. The Apache sites (A and B) used a TCP connection to retrieve advertisement from the shared FCGI, which was located on the same machine as the two sites. Since we used a TCP connection, we could have also moved the FCGI to a remote server without breaking the VS paradigm. Nevertheless, the remote server would have been so underloaded that interference effects would not have been easy to observe. Moreover, the time required to invoke a remote FCGI would have severely limited overall HTTP throughput. The load offered to Site A was kept at 30 simultaneous connections while the load offered to Site B increased from 10 to 60 simul-



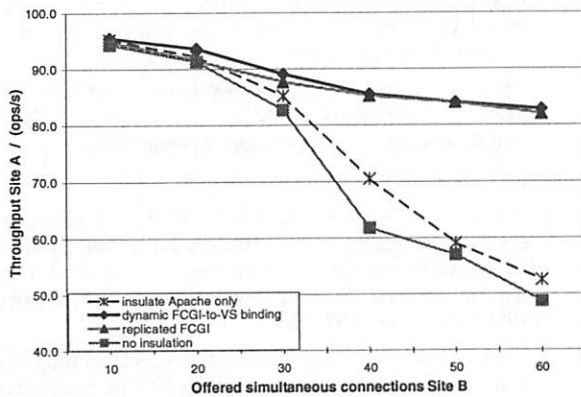


Figure 12: A's performance while increasing load on B

taneous connections — with the changed dynamic mix, the server became saturated at a total of 60 simultaneous connections of offered load.

As in the experiments of the last section, two VSs (`www1` and `www2`) were created and each was assigned half of the machine's CPU capacity. The first experiment (Apache insulation only) executes the FCGI outside the VS context of either site so that it could utilize all unused server capacity.

In the second round (dynamic FCGI-to-VS binding) we loaded the additional `accept` and `socket` gates to police access to the FCGI, which received its requests via TCP. The following classification rules were instantiated:

```
(accept, www[1|2], req = www[2|1]) → (www[2|1])
```

The `accept` rules cause the FCGI to change its resource binding if it is executing in the VS context of `www1` (`www2`) and receives a request from `www2` (`www1`) to `www1` (`www2`). Moreover, `accept` reorders requests in the order of their VS's remaining resource share as explained in Section 4. The default `socket` rule associates a new socket with its creator's VS. This ensures that requests sent to the shared FCGI will have appropriate TOS markings. To establish a baseline for optimal insulation, we replicated the FCGI script in each VS context (replicated FCGI). This cannot be done in real setups because many applications are not designed to be replicated.

As Figure 12 shows, sharing the FCGI without the `accept` gate breaks the insulation between sites A and B (Apache insulation only); the performance of Site A decreases rapidly as the load on Site B increases. This effect can be traced back to the contention for the shared FCGI. With the `accept` gate (dynamic FCGI-to-VS binding), the performance of Site A drops at a much slower, nearly the pace for replicated FCGI. The benefit of using the `accept` gate is a performance improve-

ment for the well-behaved site (well-behaved means that its clients do not overload the site) of approximately 60% under maximal load. Further experiments show that the `accept` gate for dynamic VS bindings performs almost as well as if the shared service were replicated for each VS (replicated FCGI). The ill-behaved Site B suffers from overloading its CPU share. This results in a 10% loss of aggregated performance compared to the ideal case of a replicated FCGI under peak load. The reason for this is that the ill-behaved site uses its resources mainly on serving static HTTP requests. Only when the number of queued-up FCGI requests is large will its FCGI requests be processed. During those times Site B operates mostly sequentially.

Without changing Apache this problem could not have been solved using RC's or any other approach presented in Section 2, because the resource binding for the FCGI must be dynamic, assuming it cannot be replicated.

## 6 Conclusions

We demonstrated that VSs are an effective, application-transparent resource management abstraction when sub-services are shared across business clients. Furthermore, our implementation showed that VSs can be integrated into an off-the-shelf OS without incurring much additional overhead. To manage VSs, a limited understanding of the managed applications suffices. In particular, one needs to know how services and sub-services interact. On the basis of these knowledge, the VS architecture transparently and dynamically updates the VS binding for service activities and thereby their resource bindings.

VSs are shown to be able to emulate the VH abstraction. Furthermore, we have shown that VSs provide sound insulation between competing services in spite of shared sub-services. ASPs who multiplex hardware and software resources among their business clients, benefit greatly from the proposed solution. Given the great interest in the outsourcing market, future versions of commercial resource management approaches such as WLM or Sun Resource Manager, will consider the interference caused by shared services between otherwise well-insulated services. They may use VS tracking to minimize this interference, thus improving resource management for multi-tier services significantly.

Since the VS architecture is extensible, one may choose only a small set of classification mechanisms and limited configuration options for gates. This allows a staged integration of VS tracking into off-the-shelf OS's. VS can also be integrated easily by putting the classification rules into a separate look-up table. Then a VS descriptor reduces to a RC or Reservation Domain. Therefore, it is

possible to augment RC's or Eclipse to provide VS-like dynamic resource bindings by introducing a classification table and intercepting the VS relevant system calls.

In spite of the overall acceptable performance of our experimental implementation, there is still sufficient room for improvement. To speed up classification in a commercial OS, filtering and classification should be tightly integrated into the intercepted system calls as opposed to simply placing call-back hooks inside system call code — calling an empty C function on a 300MHz AMD K6 already takes 9 $\mu$ s. A tighter integration would also avoid duplicate lookups of processes, file descriptors, etc., once to execute the system call and another time to execute classification.

The primary remaining issue in insulating co-hosted sites from each other using VSs is file cache management. To improve insulation, each disk-bound VS should be equipped with its own file cache [5]. To accomplish this goal, the inodes in the file cache must be tagged with their VS affiliation. Furthermore, one must limit the total number of inodes in the file cache for each VS. If an inode is shared by two or more VSs it should retain the tag of the highest priority VS that is using it. Otherwise, priority inversion would result. Although easy to describe, this feature requires substantial changes to the structure of the file cache. Nevertheless, content servers with very large inode working sets would benefit from such insulation. It is possible that this eliminates the small performance degradation of the well-behaved site in Figure 12.

## References

- [1] AMAN, J., EILERT, C. K., EMMES, D., YOCOM, P., AND DILLENBERGER, D. Adaptive Algorithms for Managing Distributed Data Processing Workload. *IBM Systems Journal* 36, 2 (1997), 242–283.
- [2] BANGA, G., DRUSCHEL, P., AND MOGUL, J. Resource Containers: A New Facility for Resource Management in Server Systems. In *Third Symposium on Operating Systems Design and Implementation* (February 1999), pp. 45–58.
- [3] BRUNO, J., BRUSTOLONI, J., GABBER, E., OZDEN, B., AND SILBERSCHATZ, A. Retrofitting Quality of Service into a Time-Sharing Operating System. In *USENIX Annual Technical Conference* (June 1999).
- [4] BRUNO, J., GABBER, E., OZDEN, B., AND SILBERSCHATZ, A. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *USENIX Annual Technical Conference* (New Orleans, Louisiana, U.S.A., June 1998).
- [5] CAO, P., FELTEN, E. W., AND LI, K. Implementation and Performance of Application-Controlled File Caching. In *First Symposium on Operating System Design and Implementation* (1994), ACM.
- [6] CISCO INC. Local Director (White Paper) [http://cisco.com/warp/public/cc/cisco/mkt/scale/locald/tech/lobal\\_wp.htm](http://cisco.com/warp/public/cc/cisco/mkt/scale/locald/tech/lobal_wp.htm). 2000.
- [7] ENSIM CORP. *ServerXchange (White Paper)*. Mountain View, California, 2000. [http://www.ensim.com/products/wpaper\\_fr.html](http://www.ensim.com/products/wpaper_fr.html).
- [8] F5 CORP. <http://www.f5.com/bigip/>. 2000.
- [9] GOYAL, P., GUO, X., AND VIN, H. M. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Second Symposium on Operating Systems Design and Implementations* (Seattle, Washington, U.S.A., October 1996), ACM, pp. 107–122.
- [10] HAND, S. M. Self-Paging in the Nemesis Operating System. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, February 1999), USENIX, pp. 73–86.
- [11] HYDRAWEB TECHNOLOGIES, INC. Load balancing (White Paper). [www.hydraweb.com](http://www.hydraweb.com), 1999.
- [12] JEFFAY, K., SMITH, F., MOORTHY, A., AND ANDERSON, J. Proportional Share Scheduling of Operating System Services for Real-Time Applications. In *Proceedings of the 19th IEEE Real-Time Systems Symposium* (Madrid, December 1998).
- [13] LAMPSON, B. W., AND REDELL, D. D. Experience With Processes and Monitors in Mesa. *Communications of the ACM* 23, 2 (February 1980), 106–117.
- [14] LAURIE, B., AND LAURIE, P. *Apache: The Definitive Guide*, 2nd ed. O'Reilly & Associates, February 1999.
- [15] MERCER, C., SAVAGE, S., AND TOKUDA, H. Processor Capacity Reservers: Operating System Support for Multimedia Applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems* (May 1994), IEEE.
- [16] OIKAWA, S., AND RAJKUMAR, R. Linux/RK: A Portable Resource Kernel in Linux. In *Work in Progress 19th IEEE Real-Time Systems Symposium* (Madrid, December 1998).
- [17] POSTEL, J. Internet Protocol Darpa Internet Program Protocol Specification. IETF RFC, September 1981.
- [18] SPATSCHECK, O., AND PETERSON, L. L. Defending Against Denial of Service Attacks in Scout. In *Third Symposium on Operating Systems Design and Implementation* (February 1999), pp. 59–72.
- [19] STANDARD PERFORMANCE EVALUATION CORPORATION. *SPECWeb99 (White Paper)*. <http://www.spec.org/osg/web99>.
- [20] SUN MICROSYSTEMS INC. *Solaris Resource Manager 1.0 (White Paper)*. Palo Alto, California. <http://www.sun.com/software/white-papers/wp-srm/>.
- [21] SUN MICROSYSTEMS INC. *Sun Enterprise 10000 Server: Dynamic System Domains*. Palo Alto, California. <http://www.sun.com/servers/white-papers/domains.html>.
- [22] VERMA, D. *Supporting Service Level Agreements on IP Networks*. Macmillan Technical Publishing, 1999.
- [23] WOLFF, R. W. *Stochastic Modeling and the Theory of Queues*. Prentice Hall, Englewood Cliffs, NJ, 1989.

# Location-Aware Scheduling With Minimal Infrastructure\*

John Heidemann  
USC/ISI

Dhaval Shah  
Noika

## Abstract

Mobile computers often benefit from software which adapts to their location. For example, a computer might be backed up when at the office, or the default printer might always be a nearby one. In many existing systems, location-triggered actions are only possible for specific applications or with special infrastructure. This paper describes *lcron*, a system which supports *user-configurable* actions triggered on *change in location* or other events common to mobile computers. Key features of *lcron* are its use of existing clues for location information and mapping low-level location information into user-sensible terms. *Lcron* uses a number of existing sources of location such as network connection and base station ID, allowing it to work without special hardware or GPS receivers. We map sources of low-level information such as IP address and latitude/longitude into user-meaningful *logical* locations. We describe the design, implementation and our experiences with this system.

## 1 Introduction

Laptop computers are widely used today at home, at the office, and on the road. With such computers, location-aware programs could simplify a number of common tasks. Reminders can be sent to a user when a location is reached (“feed the cat when you get home”). Location-aware queueing is easy (“print this job when I’m at work”). System defaults can be updated when a location is reached (“print to the printer in room 232 when I’m at work”). Replicated or cached data can be refreshed when convenient (“back up my hard disk when I have a good network connection”).

One of the reasons location-aware software has seen relatively little use is the perception that it requires special hardware or software. Pioneering work with the Xerox PARC TAB [11] and the ORL Active Badge system [14] took place in the context of office-wide wireless infrared networks which provide location information. These and other systems [17, 2, 4] proposed new frameworks for location-reactive software.

Lack of general-purpose location-aware software is surprising given the wide availability of portable computers and the increasing availability of wireless networking. Although some specialized applications such as mail and printing include some location awareness (perhaps queueing messages or print jobs until a network

or server becomes available), these servers are usually custom designed. As a result they may have duplicative or limited detection mechanisms and rarely support user-controlled detection and actions.

This paper examines a system which uses widely available hardware sources of location and existing software interfaces to provide user-configurable, location-triggered actions. We use wired (typically Ethernet) attachment points, but we also can use wireless base-station identifiers or GPS if available. We support easy user configuration by mapping low-level sources of location into user-meaningful tags and by expanding existing mechanisms to specify events in time to also specify events in space. Our model for general events is based on the *cron* and *at* programs widely used in Unix and recently available in Microsoft Windows with packages such as the Norton Program Scheduler and Microsoft System Agents. We believe that there is substantial benefit to adding a notion of location to user-configurable scheduling.

The main contribution of this paper are experiences with what kinds of high-level location information are relevant to mobile users, and how we can generate this information from widely available, low-level sources. In the examples above, concepts such as “work”, “home” and “fast network” may not directly correspond to sources of physical geographic location such as GPS. The triggered-event model we describe is actually more general, supporting actions triggered on arbitrary events such as current power status. Finally, we describe our experiences using this tool.

\*The authors can be contacted at 4676 Admiralty Way, Marina del Rey, CA, 90292-6695, or by electronic mail to {johnh,dhavalsh}@isi.edu. John Heidemann’s work was partially funded under the VINT project (DARPA grant ABT63-96-C-0054) and the LSAM project (DARPA grant J-FBI-95-185).



Although we see location awareness as the primary application for our work, location is actually one instance of more general *event awareness*. In addition to location, laptop users often wish to respond to other system events such as power constraints (“power is low, so shut down the wireless network and don’t scan for viruses now”). Other events may arise from the interaction of systems (“remind me to ask ‘what’s new’ when I next meet a particular colleague”). Our system generalizes to support actions triggered from arbitrary events such as these.

In a broader sense, we see this paper as one step in understanding the question raised by the Dataman project at Rutgers: “what if location were a first-class operating-system concept, much as time is now?” [6].

## 2 Using Location Information

To understand the requirements of *lcron*, this section considers what tasks might be simplified if they were triggered based on location and other events. From these tasks we generalize what kinds of location information are important to users.

In the introduction we identified a number of user tasks that would benefit from location information. We believe that many of the tasks which would benefit from location awareness require a much more refined definition of location than simple physical location as provided by GPS. Table 1 breaks these examples into several classes. Although some tasks depend on physical location, many tasks depend on hardware or some external computing capability. Other tasks should occur when another user (or other moving object) happens to be present. Finally, across each of these categories, a task could depend on a specific or a generic resource.

Some examples make these dimensions clearer. For a generic physical location, one might want to be reminded to drop off a shirt when one is near the next dry cleaners, where any cleaners is acceptable. Later one might want to be reminded to pick that shirt up when near that particular cleaners.

In today’s mobile computing environment, location aware tasks are often hardware or device specific. Printing is one example. Network access to replicate or back up data or send mail are common examples. Again, these examples may be specific (backup my computer when connected by a high-speed network to our company’s backup server) or generic (send mail when next connected to the Internet).

Some may argue that ubiquitous wireless networking may eliminate this class of jobs. Why queue mail if you’re always connected to the network? Although we believe that connectivity will reduce these needs, cost and especially power constraints may limit use of

otherwise available networking.

*Lcron* is most useful for systems which connect to several different networks, but it is also useful for connection and disconnection to a single network. Detection of network reconnection can be used to trigger events such as sending queued mail or synchronization of a disconnected file system [7].

Finally, although we have focused on location-aware job scheduling, other events map well into the *lcron* model. For example, computers often do periodic housekeeping (defragmenting a disk, checking for viruses, indexing mail); on a possibly power-constrained laptop these tasks are best scheduled when not running on battery power (when “at” the “AC-powered” virtual location). We have recently added power-triggered events and are currently investigating how useful *lcron* is for general events.

## 3 System Location

The first step in location awareness is identifying sources of location information. This section considers three potential sources of location information: the global positioning satellite (GPS) network, physical network conditions, and reports of base-stations in wireless networks.

GPS receivers provide an obvious source of location information. With recent price reductions (at the time of writing US\$100–200 receivers are not uncommon), GPS receivers seem increasingly attractive. GPS receivers work in most open areas, but they have limited use in buildings or other areas of limited reception. Unfortunately, power consumption and antenna space may be inconvenient for some portable computer users. Also, GPS accuracy is both a help and a hindrance. On one hand, GPS locations are too detailed for direct use. How many people know the latitude and longitude of their office? On the other hand, current accuracy of GPS receivers is not sufficient to place a user in the room of a building without additional processing.

For computers which are frequently networked, the wired network infrastructure itself can provide location information. Portable computers may be assigned different IP addresses and routers (or different mobile IP addresses [9]) depending on where they are. By monitoring the network attachment we can determine where the computer is. The opposite of GPS accuracy, networks attachments only vaguely specify physical location (somewhere on a given IP subnet<sup>1</sup>). However, if location-dependent jobs often require the network (for

<sup>1</sup>A reviewer suggested using ethernet bridging messages to narrow this to a specific Ethernet segment.



constraint	uniqueness	example
physical location	specific	remind me to get shirts at the cleaners
physical location	generic	what will the weather be here, tomorrow
device	specific	print this document to the printer with my letterhead
device	generic	send this message when connected to a network
other users	specific	remind me when I see a particular person
other users	generic	send the next meetings agenda to everyone in this room
other events	—	scan for viruses when not running on battery power

**Table 1: Several classes of location-aware tasks.**

example, computer backup), network attachment may be more relevant than GPS-measured physical location.

Wireless networking too can provide location information. Most wireless protocols support multiple base stations or cells. If these systems know their physical location and can report it to the *lcron*, then both network connectivity and physical location can be determined.

One important implementation issue moderates location detection. First, although one would like to detect location changes exactly, in some circumstances polling is required. Sometimes change-in-location is impossible to detect at the device level, for example with GPS where the data is continuous. In addition, there may be no operating system or driver support to trigger *cron* when a location is changed. (For example, laptops with built-in Ethernet will not generate PCCard eject events, and most Ethernet drivers do not trigger user-programs on loss of carrier.) In these cases we must poll location periodically, possibly using battery unnecessarily. Fortunately it is often possible to trigger *cron* only when location changes. For example, PCCard insertion events often correspond with network changes.

A more difficult issue is that, although these approaches work well at detecting arrival at a new location, the actions that can occur on departure from a location are more limited. Departure can be identified by lack of a network connection, but this time is too late for actions that require the network. For example, the policy of “fetch my mail just before I leave work” cannot be directly implemented. An analogous problem exists when caching data stored on removable disks, suggesting two solutions. We can either require users to inform the operating system of an impending change, or we could use computer-controlled eject mechanisms for PCcards. Since no such hardware exists currently, software approaches are required.

## 4 Mapping From System to User Location

Although we have identified a number of ways a

system will directly measure location, these approaches are often too low-level for typical users. We believe that one key to useful location awareness is a mapping from low- to user-level information, much as the domain name system maps from 32-bit IP addresses to human-friendly hostnames.

As examples of this mapping, consider GPS and network location. GPS receivers report latitude and longitude. Because of measurement accuracy, a user must specify “within 100m of this lat/lon”, instead of “at this lat/lon”. For many, lat/lon are as difficult to manage as IP addresses, so another level of mapping should allow “at ISI” or “near USC” rather than “within 100m of 33.97988N, 118.43994W”. Similar examples apply to network location. Few users remember what network segment or router they use, but “on the network I use at USC” is obvious.

Although high-level mappings simplify location description, it is also important to match location description to the task mix. “At ISI” could mean “when connected to the high-speed ISI network” to do backups, “when connected to any ISI network” to print a document, or “when physically near ISI” to send a reminder message. A flexible mapping mechanism can make these distinctions visible. We believe a larger user-base is required to understand how important these distinctions are.

## 5 *Lcron* Implementation

This section summarizes several aspects of *lcron* implementation, including how location is specified and how changes are detected.

### 5.1 Base *cron*

*Lcron* is implemented as a modification of an existing *cron* implementation and several helper programs. We based *lcron* on Geoff Kuenning’s implementation *xcron* [8]. Although other freely available *cron* implementations are more widely used, *xcron* had several features important to us for mobile use. *Xcron* is aware that

computers may be turned off or suspended; it optionally runs jobs scheduled during down-time when the system next starts. *Xcron* also includes integrated support for delayed one-shot jobs (“*at* jobs”), simplifying support for location-aware *at* services. Finally, *xcron* supports both the traditional crontab format (table driven with one column per field) and a newer format without strict columns. The newer format uses context to interpret the time specification, so “1:00” means to run at 1 a.m. daily.

## 5.2 Specifying location

Easy user configuration of event-triggered actions is a goal of *lcron*. We accomplish this by adding an “event” field to the existing crontab file format. A user’s crontab file lists the commands that are to be executed on user’s behalf at specified times on specified dates. The new field specifies that events should be executed when a particular location is reached or a particular event occurs. Actions can be triggered either periodically when at a location, or only when that location is first reached.

A sample crontab with location-triggered jobs appears in Figure 1. The optional location field begins with the “@” sign. In this example, we fetch mail 3 times a day when at home but every 20 minutes when at work and when we first connect to the work network; these entries have both time and location specifications. Other events are triggered only when we first arrive at location. The “m” flag and lack of a time specification indicates this behavior. An example is setting the default printer to vary depending on where we are. Finally, when we connect to the work network and are powered we run a program to back up the portable computer. (We have implemented and regularly use all but the ability to “and” together multiple events.)

We have also modified *xcron*’s *at* program to provide a similar event specification. Users can schedule jobs “at 8pm @home”.

## 5.3 Location sources and detection

Our primary source of location information is network connectivity. We examined GPS receivers as an additional source of information, but most location-aware applications we wanted to schedule depended on network connectivity, so GPS receiver cost, size, and power requirements limited its use. We also map battery power into *lcron* as an example non-location events.

To sense network location we measure what networks are currently configured and map either the gateway host or the network IP address and mask to a user-sensible location (as described in the next section). Currently we sense network attachments with the “netstat -rn” command. Hosts with dynamically assigned IP

addresses (with DHCP, for example) may not have stable addresses, so we primarily use gateway addresses to identify location.

We can detect location changes both by *polling* and *triggered notification*. Triggered notification avoids the delay and constant (if low) overhead of polling. We use triggering as our primary means of detecting discrete locations. Over time we have used two ways to detect change in our system (a laptop running the RedHat distribution of Linux). Originally we detected change to network connectivity with a one-line addition to the PCCard- and PPP-configuration scripts that are used with our system. This line informs *cron* of location change after a new network is started. More recently (since RedHat 6.0) we have used the operating system’s built-in ability to signal jobs when the network changes.

We also implement polling as a secondary mechanism to detect location changes. To implement polling, a non-location-specific *cron* job periodically checks the system’s location, noticing and acting on any changes. The polling interval can be selected to trade off responsiveness and overhead; by default we poll every 10 minutes. Polling is required for events which are continuous in value, such as GPS data and battery power. Polling is disabled if continuous events are not needed and triggering is possible.

We recently added support to detect power transitions (between AC and battery power) as an event. Ric Faith and Avery Pennarun’s *apm* daemon triggers a program when power mode changes. We wrote a small stub that maps these events in to *lcron* events. Support for different classes of events like this motivates support for the ability to “and” events together (when powered and network connected).

## 5.4 Mapping from system to user

In addition to specifying which actions are triggered in response to particular events, users should also be able to specify how location is defined. As discussed in Section 4, system-measured location information is not always appropriate for direct user consumption. *Lcron* therefore employs a mapping function from system to user locations.

We have experimented with two ways to implement this mapping. Originally mapping was done with a short Perl program which filters raw locations into user-sensible ones. We chose to implement mapping as a program rather than through a table to increase flexibility. Networks might map gateway names into locations, while GPS data might be mapped based on physical proximity or more detailed topological understanding.

Since currently these capabilities are not widely used, we have replaced it with a table driven mapping (from

```

m          @net:home set_default_printer home_lp
- 7,13,20:00 @net:home fetchmail
m          @net:work set_default_printer ps11d_d
- :0-59/20   @net:work fetchmail
m          @net:work fetchmail
m          @power:battery enable_hard_disk_spindown
m          @power:ac      disable_hard_disk_spindown
m          @net:work&&power:ac backup_disk

```

**Figure 1: A sample crontab specification with location-specific commands.** (Unlike standard Unix *cron*, *lcron*'s first field is list of flags, defaulted fields may be omitted, and the @ field specifies location constraints.)

```

128.9.160.7      isi
128.9.128.3      isi-netwave
128.125.187.254  usc
128.9.97.33      home
128.9.32.13      ppp
128.9.176.100    ppp

```

**Figure 2: A mapping table from gateways to user-level locations.**

gateway to logical name) that is easier to configure. Figure 2 shows a sample mapping and illustrates that in some cases multiple low-level locations may map to the same logical location (ppp, in this case). Since mapping is user-dependent we plan on a simple tool which associates the current system location (based on one or more low-level criteria) with a user location.

## 6 Experiences using *lcron*

*Lcron* has been used by a few researchers at ISI since January 1998. Our users have different levels of network connectivity. At the low-end, one user's laptop attaches to a single network. At the other end, one user regularly uses four networks in three different physical locations (ISI, USC, and home). This section briefly describes our experiences developing and using *lcron*.

### 6.1 User environment

In Section 2 we described and classified a number of location-dependent tasks. We have considered *lcron* for four classes of day-to-day tasks:

- File replication and e-mail transfer
- Configuration of system environment
- Alarm service
- User-interface teleporting [10]

We currently use *lcron* for first three tasks on a daily basis. We have also experimented with user-interface teleporting (automatically bringing up a copy of running applications on a nearby display).

None of these applications are new: file-replication [7, 5], automatic system configuration [3, 16, 12], location-specific alarms [16, 12], and teleporting [10] have been experimented with before. The advantage of *lcron* is that now these applications can be easily deployed by users without the need for any extensive supporting infrastructure.

A common and effective use of *lcron* is for file replication and e-mail transfer. Since e-mail is often timely, it is helpful to immediately send or fetch queued messages upon connection. In addition, periodic e-mail retrieval is easily configurable with *lcron*. User-customization is helpful here; the importance of timely e-mail delivery can be weighed against battery constraints (when connected by a wireless network) and tolls (when connected from home via a metered service). This level of configuration would be difficult or impossible with the simpler retrieval models of most typical e-mail packages.

File replication poses similar problems. Automatic file backup is important to a safe environment. *Lcron* triggers this backup periodically when connected to a fast network.

For both e-mail and file backup we had previously employed customized systems of polling. To avoid draining laptop power we previously polled for the laptop from a server machine. Although functional, this system was not easy to change and required configuration on multiple machines. Replacing this system with *lcron* simplified configuration and concentrated it on the laptop under user control. As a result of these simplifications we automated cases that before were not considered important enough to justify the effort (for example, occasional but automatic retrieval of mail from secondary mailboxes), or were not feasible (immediately fetching mail on connection).

Examples of location-dependent system configuration are selecting a default printer and telephone dialing. We



wrote a small program which changes the definition of a particular printer entry when the laptop arrives at a new location. Users can thus set their default printer to the printer called “nearest” to print to a location-dependent nearby printer. Similarly different locations have different context for telephone dialing (area codes, handling of extensions, and telephone interface). Our telephone auto-dialer supports these options; *lcron* allows us to automatically select between them as we do for printers. Other location-dependent tasks similar to these can be completed automatically using *lcron*.

A third example application we experimented with is a location-based notification service. Users can record “reminder messages” which are sent as e-mail when a given location or time is reached. In other cases delayed actions have proven useful (for example, downloading a file when next connected to the net).

We also experimented with teleporting in *lcron*. First done in the VNC system [10], teleporting is the idea that a user’s existing applications should move transparently to the nearest, most capable display. We have experimented both with VNC teleporting and a weaker form where new sessions of a set of applications are automatically begun on a nearby display when a network connection is made. Both have been useful, although automatic teleporting is not always desirable (for example, when the laptop is only briefly connected to send and receive mail).

From our experiments with *lcron* the primary benefit is the automation of day-to-day tasks that were difficult or not warranted beforehand. In some cases of e-mail retrieval *lcron* replaced manual requests. In others it replaced older, less capable and more complicated retrieval mechanisms based on application-specific polling. Finally, the simplicity of *lcron* configuration supported additional uses. Prior arrangements required configuration on multiple machines to avoid polling from the power-constrained laptop; *lcron* simplified configuration by allowing all configuration to take place on the laptop.

## 6.2 Development environment

*Lcron* was developed under SunOS and several versions of RedHat Linux. Based on *xcron*, it inherits that system’s portability. Our primary platform has been laptops running various versions of RedHat Linux. Other than small OS changes to collect location information are specific to RedHat Linux, the core of *lcron* is portable to any Unix platform.

In early versions of RedHat Linux, small changes to the base operating system’s network configuration scripts are required to allow *lcron* to avoid polling. We made 4 lines of modification to RedHat Linux’s PCCard

and PPP scripts (no other OS changes were required). These are not necessary with more recent versions (since RedHat 6.0).

We found one hardware limitation: we detect network configuration based on PCCard insertion and removal. One laptop included a built-in Ethernet adaptor that lacked these events. Polling the network for carrier can work around this problem; another approach would be to modify the network driver to allow an application to detect changes in the Ethernet carrier.

Our experiences with *lcron* development suggest that it should be easily portable to other versions of Unix. Periodic polling (perhaps at 5 minute intervals) can be used to detect network change, but we expect that direct detection with small changes will often be a possible optimization.

## 7 Related work

*Lcron* builds on three areas of related work. First, groups such as the Dataman project, Xerox PARC, and the Olivetti and Oracle Research Lab have looked at general ways location awareness changes system behavior. Second, a number of groups have looked at how to modify actions to consider location information. Finally, several special purpose systems have developed custom approaches to watch for location changes.

Our work was inspired by the Dataman project’s question of “how would system software change if location were a first-class operating-system concept” [6]. The Dataman project has looked at how mobility affects network transport (mobile IP) and multicast (geographic messaging). We apply their proposition in a very real sense by considering the use of location in existing approaches to task scheduling.

Xerox PARC’s work in ubiquitous computing [16] through systems such as the PARCTAB [15] pioneered location-aware computing. Their work in context-aware computing described systems similar in function to *lcron* [12, 11]. *Lcron* builds upon this work by describing how multiple, commonly available of location information (such as network connections and GPS information) can be mapped to user-relevant locations.

Schilit, Adams, and Want classify context-aware computing along two axes (see Figure 2 from [11]) based on whether the task at hand is performing information retrieval or command execution and whether it is done manually or automatically. By this classification *lcron* is automatic (not manual) and supports commands (not information), thus falling into the “context-triggered actions” quadrant. However, we have shown how context-triggered actions in *lcron* allows us to implement contextual commands (for example, printing to the



	manual	automatic
information	proximate selection/contextual info	auto contextual reconfig
command	contextual commands	contextual-triggered actions

Table 2: Schilit, Adams, and Want's classification of context-aware computing (from [11]).

nearest printer) and automatic contextual reconfiguration (for example, by automatically replicating data), thus providing some support for some kinds of context-aware applications which are not directly supported. We believe that the final quadrant (manual information retrieval) is best solved with other work such as location-aware web-browsers [1, 13].

Location dependent information such as nearest restaurant, availability of space in the nearest parking lot or local weather information can best be retrieved on-demand by location-aware web-browsers. On the contrary *lcron* mainly targets those location dependent tasks whose timely execution on location change is important for proper functioning of the system. However *lcron* could also be used for automatic information retrieval by scheduling an information retrieval task to be executed at different locations. *lcron* thus gives users explicit control over the nature and contents of location-dependent information to be retrieved in contrast to the limited set of information available to the users of location-aware web browser.

## 8 Future work

There are several directions for future work. Our primary sources of location come from network attachments; additional experience with other sources would be helpful. Exposing the cells of a cellular network might be attractive since that information is already available. Wider use of GPS information would also be useful.

Improvements in mapping arbitrary events into *lcron* would also be helpful. We are currently developing and experimenting with mapping changes in power consumption to events; more experience here would be helpful.

Currently *lcron* focuses on one variable: what happens as the system changes state by moving around or losing power. Some location-aware applications have more sophisticated requirements. For example, reminders could be triggered when two people are co-located assumes multiple moving objects, or actions triggered when at a location and AC-powered. Work in active badges suggests that these kinds of interactions can be provided by periodically broadcasting presence information to the local area. Refinement of this idea in *lcron* remains future work.

*Lcron* currently uses a table-driven approach to specify event triggers. This tabular approach is easy for users but can limit specification flexibility. For example, it's easy to take actions at particular times if connected, but not easy to schedule a job at a connection and then at regular intervals after that time. Whether the limitations of this fairly rigid structure is a problem in many situations remains to be seen. We work around this problem with hourly polling through a helper program. Possibly a better solution is to specify events triggers through a programming language (as we map from system to user locations) and to use a front-end to construct program statements for simple cases.

## 9 Conclusions

We have described *lcron*, a system supporting user-configurable actions in response to location- and event-triggers. It has two key features: first, it uses existing sources of information, such as wired and wireless networking information, to determine location. Second, its approach at mapping these low-level sources of location and other events to user-sensible terms. Our experiences with *lcron* suggest that it achieves its goal of simplifying user-triggered actions in response to location changes and other events, and that this simplification allows much broader automation of location-triggered actions than alternatives.

## Acknowledgments

The authors thank Geoff Kuenning development of *xcron*, support for integrating *lcron* and discussions of *lcron*-related issues. We would also like to thank Ramesh Govindan, Joseph Bannister and the anonymous reviewers for careful readings and suggestions to improve this paper.

## Software Availability

We plan to make the software developed as part of this paper publicly available. Please contact the authors for current status.

## References

- [1] Arup Acharya, B. R. Badrinath, Tomasz Imielinski, and Julio C. Navas. A WWW-based location-dependent information service for mobile clients. At [http://www.cs.rutgers.edu/~navas/dataman/papers/loc\\_dep\\_mosaic/Overview%.html](http://www.cs.rutgers.edu/~navas/dataman/papers/loc_dep_mosaic/Overview%.html), July 1995.
- [2] H.P.W. Beadle, G.Q. Maguire, Jr., and M.T. Smith. Using location and environment awareness in mobile communications. In *Proceedings of the IEEE/IEEE International Conference on Information, Communications, and Signal Processing*, pages 1781–1785, Singapore, September 1997. IEEE.
- [3] Mic Bowman, Larry L. Peterson, and Andrey Yeatts. Unvers: An attribute-based name server. *Software—Practice and Experience*, 20(4):403–424, April 1990.
- [4] P. J. Brown, J. D. Bovey, and X. Chen. Context-aware applications: from the laboratory to the marketplace. *IEEE Personal Communications Magazine*, 4(5):58–64, October 1997.
- [5] John S. Heidemann, Thomas W. Page, Jr., Richard G. Guy, and Gerald J. Popek. Primarily disconnected operation: Experiences with Ficus. In *Proceedings of the Second Workshop on Management of Replicated Data*, pages 2–5. University of California, Los Angeles, IEEE, November 1992.
- [6] Tomasz Imielinski and B. R. Badrinath. Wireless mobile computing: Challenges in data management. *Communications of the ACM*, 37(10):18–28, October 1994.
- [7] James J. Kistler and Mahadev Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.
- [8] Geoff Kuenning. Experiences with an extended cron daemon. Unpublished manuscript, November 1997.
- [9] C. Perkins. IP mobility support. RFC 2002, Internet Request For Comments, October 1996.
- [10] Tristan Richardson, Frazer Bennett, Glenford Mapp, and Andy Hopper. Teleporting in an X window system environment. *IEEE Personal Communications Magazine*, 1(3):6–12, July 1994.
- [11] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 85–90, Santa Cruz, CA, USA, December 1994. IEEE.
- [12] Bill Schilit, Marvin Theimer, and Brent Welch. Customizing mobile applications. In *Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing*, pages 129–138, Cambridge, MA, USA, August 1993. USENIX.
- [13] Geoffrey M. Voelker and Brian N. Bershad. Mobi-saic: An information system for a mobile wireless computing environment. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 185–190, Santa Cruz, CA, USA, December 1995. IEEE.
- [14] Roy Want, Andy Hopper, Veronica Falcao, and Jonathon Gibbons. The active badge location system. *ACM Transactions on Information Systems*, 10(1):91–102, January 1992.
- [15] Roy Want, Bill N. Schilit, Norman I. Adams, Rich Gold, Karin Petersen, David Goldberg, John R. Ellis, and Mark Weiser. An overview of the PARCTAB ubiquitous computing experiment. *IEEE Personal Communications Magazine*, 2(6):28–43, December 1995.
- [16] Mark Weiser. Some computer science problems in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, July 1993.
- [17] Girish Welling and B. R. Badrinath. A framework for environment aware applications. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 384–391, Baltimore, Maryland, May 1997. IEEE.

## Distributed Computing: Moving from CGI to CORBA

James FitzGibbon

*TargetNet.com Inc.*

Tim Strike

*TargetNet.com Inc.*

### Abstract

In this paper, we document the evolution of a banner ad delivery system from a simple CGI script written in Perl running on a single host into a distributed computing application using CORBA.

While CORBA has an established history in the enterprise-computing world, it is only recently that the OpenSource® community has begun to embrace it. Starting without any RPC programming experience, it took TargetNet a little less than half a year to integrate CORBA into the Apache web server and convert all their CGI programs into CORBA servers.

Performance of the system increased from 50 transactions per second to over 400 per second. Thanks to the cross-platform capabilities of CORBA, future components can be developed on virtually any operating system and programming language. By adding inexpensive servers, the capacity of the system scales in a near-linear fashion. Most importantly, the switch to CORBA didn't require a change of operating system or development environment – everything runs on a free operating system using OpenSource components.

### Introduction

In 1997, TargetNet built and deployed “The Datacom Ad Network”, a banner ad delivery system. The CGI script that selected and delivered ads was written in Perl, ran on a Pentium 200 machine, and could deliver just one ad per second. Rewriting in C and migration to faster processors took performance to 10 ads per second, then 30, then 50. Further optimization proved futile: we had reached the CGI performance barrier.

Increasing performance by adding processors or hosts was not feasible: the architecture of the existing delivery system was limited to running on a single host, and was single-threaded. Worse, the code used a large number of flat files on disk, and so spent a large percentage of its time performing system calls or waiting for file locks. Increasing processor power provided a brief respite, but we could not afford to upgrade server hardware forever.

To remain competitive, ad delivery performance needed to increase to roughly 400 ads per second plus allow for multiple hosts to share the load of the network. It was clear that a new architecture was required that could overcome the limitations of standard CGI scripts. The base requirements of the new system were laid down before any research began. The new system would have to:

- offer single host performance several times that of the existing CGI script.
- utilize a distributed computing model without arbitrary limits on the number of hosts.
- allow multiple hosts to share the network load, preferably with load balancing and redundancy.
- be called from a web browser like a CGI script but without the inherent limitations of CGI.
- scale to handle anticipated growth over the next four to five years without major architectural changes.
- allow gradual integration of commercial hardware and software without massive re-coding.

Most importantly, the system had to be cost effective. Our server platform was Apache running on FreeBSD, and all software in use was either freely available or developed in-house. While commercial solutions to our problem existed (Oracle Parallel Server running on a commercial UNIX), financial constraints dictated that we find a free solution or develop one in-house.

## Breaking the CGI speed barrier

The problem of CGI performance is not new. Over the last several years, many solutions that remove the forked-CGI bottleneck from web applications have come to light.

CGI enhancement wrappers like [FASTCGI] allow an existing CGI script (especially those written in interpreted languages like Perl) to run much faster and remain resident between invocations. Still, these wrappers extend the existing CGI standard, sacrificing flexibility for compatibility. They are limited, naturally, to tasks that you would usually use a CGI script for. Other client/server tasks need to be addressed separately.

Integrating the functionality of a CGI script directly into the web server provides the benefits of a CGI wrapper, and gives developers access to the internals of the web server. Not only does this technique share the same limitations as CGI wrapper toolkits, but developers have to lock themselves into a particular web server architecture, choosing to develop an ISAPI, NSAPI, or Apache module. Similarities between these architectures are few: moving a complex module from one web server product to another could require a complete rewrite.

Clustered computing solutions promise transparent scalability simply by adding hosts. Unfortunately, most clustering systems are commercial, costly, and tied to a particular line of hardware (though there are alternatives, such as Beowulf clusters running on Linux). Using commodity hardware would have addressed the cost issue, but would have required us to switch from FreeBSD to Linux. In doing so we would be giving up the significant investment we already had in FreeBSD servers and knowledgeable personnel. In short, we felt that the immediate benefits of clustering were outweighed by the commitment that one has to make to a particular vendor and/or operating system.

Remote Procedure Call (RPC) solutions do not share the aforementioned limitations. Most RPC solutions are available for multiple operating systems and hardware. They are abstracted from the application layer, and do not require adherence to one vendor's API. As it is not directly tied to the CGI model, RPC can be used to replace traditional client/server applications as well. The only issue surrounding RPC is which architecture to use.

ONC RPC, developed by Sun Microsystems, is already in wide use on UNIX® systems. ONC RPC is at the

heart of NIS, NIS+ and NFS. The limitations of the original ONC RPC have been documented and exploited for as long as they have been in use. ONC RPC+ addresses many of these limitations and provides for encrypted communication but is not as widely available as the original implementation.

The Distributed Computing Environment (DCE) from the Open Group provides almost every distributed computing tool one could need, but is as complex as it is complete. Suited best for large projects, the administration of DCE can be a monumental task. Only one free implementation of DCE is available, limiting improvement through vendor competition.

We found many of the elements we required in RPC, but we did not find an implementation that provided all of them in one package. We attempted to build our own middleware, without much success. The issues that undoubtedly plagued the developers of ONC RPC and DCE proved too much for our small team of developers. Returning to the research arena, we began to look at CORBA.

We had dismissed CORBA early in the design phase, believing it to be geared towards enterprise computing and unsuitable for our use. An in-depth examination showed promise. CORBA offered everything that we were looking for: unlimited cross-platform capability, several free implementations, and an aggressive development model that promises to keep the technology alive in the future. As discussed in [MODZ97], CORBA also offers features not found in RPC or DCE,

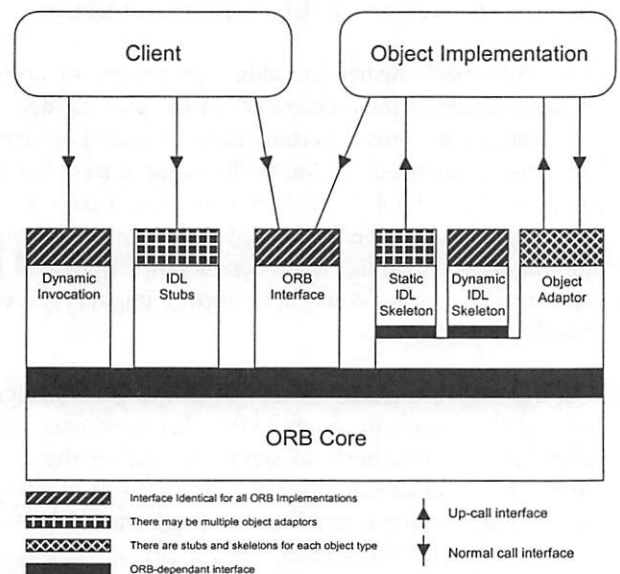


Figure 1: the CORBA Infrastructure



including interface portability, dynamic interface invocation and platform independent data types. Only one question remained: how easy would CORBA be to integrate into a system built solely from OpenSource software and in-house code?

## Back to School

The first order of business was to learn about CORBA. Starting with no more than a conceptual understanding of RPC mechanisms, we needed to deliver a proof-of-concept application in short order. To learn about CORBA, the best place to start is the Object Management Group[OMG]. The OMG produces the CORBA specification, but unlike ONC RPC and DCE, does not provide an implementation. This distinction prevents the specification group from monopolizing the implementation.

CORBA allows applications to communicate with each other, without regard to where they are located, the operating system they run on, or the programming language they were developed in. CORBA allows the developer to concentrate on the function that the application performs as opposed to the mundane details of protocol design and network transport. At the heart of a CORBA application is the ORB, or Object Request Broker. The ORB is responsible for intercepting client calls for remote methods, locating the host that provides the implementation of that method and the packaging of parameters and return values over the wire. The OMG also specifies the Internet Inter-Orb Protocol (IIOP), which is protocol that ORBs use to talk to each other. Figure 1 (from the OMG's web site) illustrates the architecture.

CORBA allows clients and servers to be written in any programming language for which a CORBA mapping has been defined. To create a CORBA server, the procedures and methods of the application are described using the Interface Description Language (IDL), which is roughly analogous to a C++ class definition. The OMG specifies both the syntax of IDL and the mapping from IDL data types to language specific data types. Presently, there are language mappings for C, C++, Ada, COBOL, Java, Smalltalk and Python. There are several independent language mappings for Perl, though they have yet to be ratified by the OMG.

An IDL compiler (part of the CORBA development environment) is used to create language-specific code from an IDL definition by generating client-side stubs and server-side skeletons. When writing a server, you add your implementation code to the generated skeletons. When writing a client, you call the functions from the generated client stubs. In either case, the finished binary is linked against a CORBA runtime object, usually in the form of a shared library or DLL.

## Choosing an ORB

At the onset of this project, TargetNet's server base consisted of FreeBSD machines running on Intel hardware. The number of freely available ORBs is significantly lower than those available for commercial UNIX systems or for Windows.

Our ideal ORB would support both C and C++, use POSIX threads for maximum server performance, and support the latest CORBA specification. We evaluated the five ORBs that were readily available for FreeBSD:

Table 1: ORBs available for FreeBSD

ORB	Supported languages	True CORBA ORB?	Performance <sup>1</sup>	License Type / Cost	Comments
mico	C++	yes	0.1 ops/sec	GPL	slow performance in testing
ILU	Many	no	not tested <sup>2</sup>	BSD-like; free	reads IDL
ORBacus	C++, Java	yes	not tested <sup>3</sup>	\$3000 per seat	SSL support
ORBit	C, C++ coming	yes	20 ops/sec	GPL	basis for GNOME
omniORB	C++, Python	yes	1250 ops/sec	GPL	multithreaded servant

<sup>1</sup> We measured performance by invoking a "null" function which took no parameters and returned no output

<sup>2</sup> The feature set of ILU did not meet our criteria, and was removed as a candidate prior to testing.

<sup>3</sup> We deferred testing of ORBacus due to its cost; the performance of omniORB finalized our decision before testing of ORBacus became necessary.

- mico [MICO], which supported only C++.
- ILU [ILU], which supported more languages than the OMG has language mappings. ILU is not a true CORBA ORB, but it is conceptually similar and can read IDL definitions.
- ORBacus [ORBACUS], which supports C++ and Java and provides an SSL interface, but is not freely available.
- ORBit [ORBIT], which is the technology at the heart of the GNOME project. ORBit supports only C, but a C++ binding is being developed.
- omniORB [OMNIORB], which supports C++ and Python.

The results of our evaluation are summarized in Table 1.

Our primary concern was standards conformance, followed by performance. Inexperienced as we were with CORBA, we didn't want to complicate matters with a non-compliant ORB. ILU failed the conformance portion early on - though it understands IDL, the language mapping is not the same as the CORBA specification.

Our test suite consisted of a simple function that took no parameters and returned no output (the intent was to measure ORB overhead). In retrospect, further research into existing benchmarks would have been beneficial; [SCHM96] describes modifications to the popular ttp network benchmark program to test CORBA performance.

mico's performance was extremely slow, taking up to 10 seconds for each transaction. ORBit worked well as a client, but its server implementation serialized remote requests, restricting performance to 20 transactions per second.

omniORB took first place for performance - its server implementation is fully multithreaded, creating a new thread for each remote request while using several "housekeeping" threads for connection management. omniORB was able to handle 1,250 transactions per second. The only downside to omniORB was that it did not provide a C language binding.

Having found a free ORB that met our performance criteria, we did not test the performance of ORBacus. Its US \$3000 price per developer scared us away; we

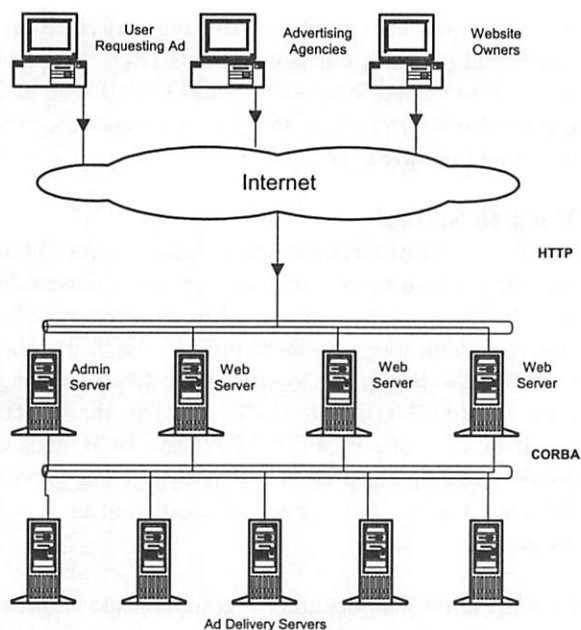


Figure 2: Datacom Mk2 Overview

may take another look at it if we require the extra features it provides.

Having completed our tests, we were unable to find all of the desired features in one product. In the end, we chose to use two ORBs: omniORB for C++ client and server development, and ORBit for C client development. Due to the performance limitations of the ORBit server model, we decided to restrict server development to C++. With our development tools in hand, we began the design and planning of the new ad delivery system.

## Datacom Mk2

Our original vision for the new ad delivery network is outlined in figure 2. There are two distinct groups of users who interact with the system over the Internet. The first group includes clients (advertising agencies) and members (the owners of the web sites that display our ads), who log into the administrative server to query how many hits and clicks have been recorded. The second group is the users who request ads as they visit member web sites. These users need to contact one of several ad servers, which select an appropriate ad to display based upon targeting criteria, and return that ad to the user.

As figure 2 illustrates, CORBA would only be used for communication between the administrative server and the ad servers. The main drawback to this model is that it requires an HTTP daemon to be installed and configured on each ad server, and that each ad server needs to be directly connected to the Internet. This precludes

the ad servers using services that are not safe to run on a public network, including file sharing or an RDBMS. In the event that an ad server goes down, users who continue to use a cached IP address will see broken images when they attempt to connect.

Our solution was to abstract the ad server communication with the Internet. Rather than talking directly to web browsers, the ad delivery machines would use CORBA to communicate with a proxy machine. The remote users would use HTTP to talk to the proxy machine, which could select the best ad server to handle the request based upon server load.

### An HTTP to CORBA Proxy

To allow a server to interact with a user on the Internet without being directly connected to the Internet, the server must make use of a proxy or gateway of some type. A firewall is an example of a proxy with which most people are familiar. Most firewalls perform simple address translation without regard to the protocol that the IP packets carry. What Datacom called for was a proxy server that could convert an HTTP request into a CORBA call, then take the result, if any, and convert it back into an HTTP response.

Using such a proxy (which we named the “Dispatch Server”) allowed us to connect the dispatch machines to the Internet, leaving all other hosts connected to a private LAN only. Converting from HTTP to CORBA was only the initial goal. With careful planning, the dispatch server could act as a gateway between other protocols as well. Future goals include gating between HTTP and a Generic NQS queue [GNQS] or an Enterprise JavaBean gateway[JAVABEAN].

Figure 3 illustrates how the dispatch server integrates into Datacom Mk 2.

The dispatch server is implemented as an Apache module, written in C, using ORBit as its ORB. When a web browser requests an ad, it contacts the dispatch server

and requests a URL such as

```
http://dispatch.TargetNet.com?target=ad&dir=bi&member=jfitz
```

This URL represents a request for a bi-directional communication with the ad delivery service, passing the single key-value pair “member=jfitz” to the server. To service this request, the dispatch server makes a connection to the ad delivery server and invokes a remote procedure named “DispatchRequest”. The name-value pairs are passed as parameters. The response from the procedure is returned to the web browser as if it had come from a CGI script. The web browser is not aware that the response it receives back from the dispatch server was actually generated on a different machine. Thanks to the cross-platform capabilities of CORBA, it does not even know if the response was generated on a UNIX machine in the same data center or on a Windows NT box halfway around the world.

For requests that do not require a response to be returned to the client, the dispatch server calls the remote procedure “Dispatch::unidirectional”, which takes the same input parameters but returns no output.

### Abstracting the Server interface

Earlier, we mentioned that for a client to invoke a remote CORBA call, it must be linked with the client stub code generated by the IDL compiler. Imagine for a moment that the ad delivery network expands such that there are fifteen different types of servers in the system. If each server defines a separate interface then the dispatch server would have to link in fifteen different client stubs in order to dispatch requests to all the servers. Every time a new type of server was added, the dispatch server would have to be re-compiled and deployed across the network. This limits the scalability of the dispatch architecture.

There are two techniques that avoid this. The first uses the Dynamic Invocation Interface (DII) features of

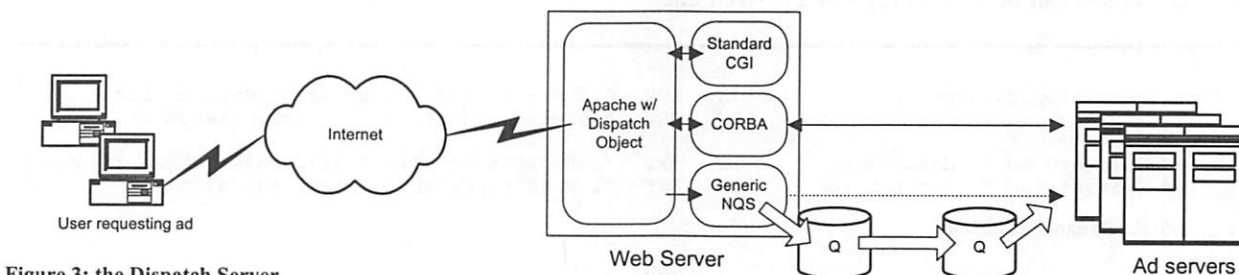


Figure 3: the Dispatch Server

```

Module Dispatch {
    typedef sequence<octet> stream;

    interface unidirectional : Control {
        long DispatchRequest(
            in string id,
            in string indata,
            in string inattr
        );
    };

    interface bidirectional : Control {
        long DispatchRequest(
            in string id,
            in string indata,
            in string inattr,
            out stream outdata,
            out string outattr
        );
    };
};

```

Figure 4: the Dispatch IDL

CORBA. DII allows a client to create references to a server at run time instead of at compile time. This is an elegant approach, but it is an advanced CORBA topic that we were (at the time) uncomfortable using. To invoke operations on arbitrary servers while still using static invocation, we chose to define a base interface from which all servers would inherit.

Recall that IDL is conceptually similar to a C++ class declaration. An interface can inherit from another interface and methods in a derived interface can override methods from their base interface. Multiple interfaces may be grouped into a module. The definition of the Dispatch interface is in Figure 4.

The Dispatch module contains two interfaces: unidirectional and bi-directional. Each of these interfaces defines one member function called DispatchRequest. Each of the CORBA servers has its own interface, which is derived from the unidirectional or bi-directional interface. As in C++, a reference to a base interface object can be used to refer to a derived inter-

face object. This allows the dispatch server to invoke the DispatchRequest function on the remote server without actually knowing which type of server it is talking to. As we become more experienced with CORBA, we may abandon this method in favor of DII. For now, our inherited base-interface technique provides the functionality we need with a minimum of complexity.

## Locating remote servers

One important step that we have not discussed is the method by which an ORB determines the specific server that provides a given interface. Most ORB implementations assign an ephemeral listening port upon server startup. For the ORB to make a connection to a remote host, it needs to know the IP address of the host, the port number and the name of the interface.

To represent this information in a platform-independent way, CORBA uses an Interoperable Object Reference, or IOR. The IOR is a text string that encodes the values of the host, port and interface for a given server. Upon server start-up, the ORB generates an IOR. The client ORB takes this IOR, decodes the values, and establishes a connection to the remote host. How, then, is the IOR communicated between the server and the client?

The CORBA spec includes interfaces to CORBA helper applications. One of these is the CosNaming service, which is a directory service that allows an ORB to register and lookup IORs. The CosNaming service provides an effective method for looking up IORs but requires that all servers use a single host as their naming service. Neither omniORB nor ORBit provides a facility for the naming service on one machine to share information with the naming service on another machine. If the machine providing the naming service goes down, servers will not be able to register their IORs, nor will clients be able to look up IORs.

There are efforts underway to replace the CosNaming service with other directory services, such as LDAP.

---

ad-delivery.iiop.datacom.rpc	IN	SRV	0 0 0 ior.ad-delivery.ad-01.datacom.rpc
	IN	SRV	0 0 0 ior.ad-delivery.ad-02.datacom.rpc
ior.ad-delivery.ad-01.datacom.rpc	IN	TXT	"IOR:72616c2d30312e746f722e646174"...
ior.ad-delivery.ad-02.datacom.rpc	IN	TXT	"IOR:6d2f52656369657665723a312e30"...

Figure 5: the Dynamic DNS zone



[RFC2714] describes an LDAP schema for storing IORs, which would eliminate many of the problems with the CosNaming service. This solution still has a single point of failure, as all updates must be made to the LDAP directory master. Rather than wait for these efforts to bear fruit, we developed our own IOR directory, building in load balancing, replication and redundancy. We call this directory daemon the Service HeartBeat Daemon.

### The Service HeartBeat Daemon

The Service Heartbeat Daemon, or HBD, performs several important tasks:

- accepts registrations from servers and keeps them in an internal table
- performs regular checks on all recognized servers to make sure that they are answering requests
- synchronizes its internal table with those of the HBDs running on other hosts.
- synchronizes a DNS zone with its internal table using Dynamic DNS updates.
- listens on a socket for clients wishing to know the IOR for a given service.

The architecture of the Service HeartBeat Daemon is conceptually similar to that of IGOR [MODZ97], though we were not at the time aware of this work. Unlike IGOR, the HeartBeat daemon does not store object references in persistent storage; rather it relies on a peer network in which daemons on different hosts keep each other up-to-date.

Upon start-up, a server determines its IOR, which is then sent to the HBD running on the local host. The HBD performs a quick check to ensure that the server is really up, echoes the registration information to other HBDs using multicast IP and then inserts the registration information into its internal table. The HBD regularly synchronizes its internal table with a DNS zone using Dynamic DNS updates. SRV records in DNS represent servers which are up and the IORs are stored using TXT records. If ad delivery servers were started on the hosts ad-01.targetnet.com and ad-02.targetnet.com, the DNS zone would resemble that in figure 5.

A client can find the IOR for a service using two methods: DNS or local socket lookup. To use DNS, the client asks a nameserver for the SRV records for a

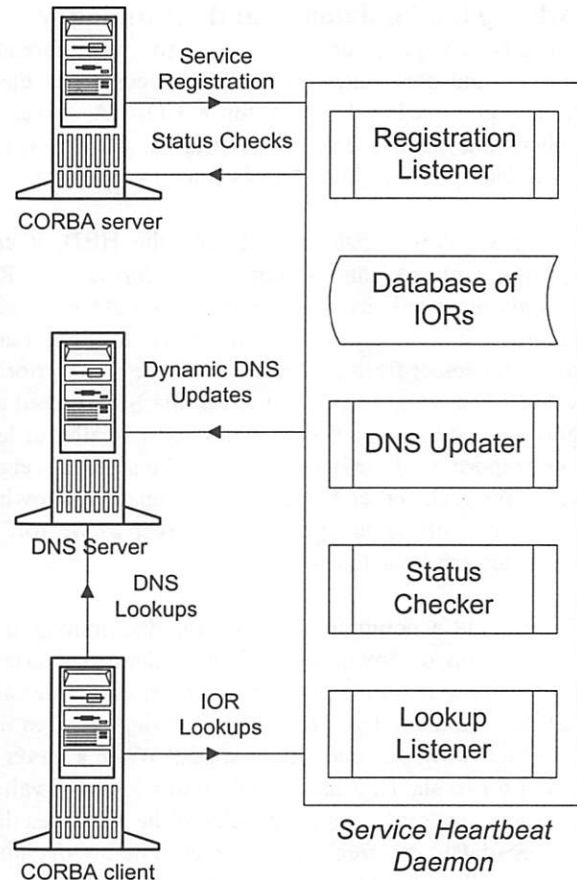


Figure 6: the Service HeartBeat Daemon

given service name. The client then uses the algorithm described in [RFC2052] to select among the multiple records returned. We deviate slightly from the RFC in that the target field of the SRV record is not the host on which the service can be found but the hostname whose TXT record holds the IOR to be used.

The DNS method of lookup has two shortcomings: it requires that the HBD keep a DNS server synchronized with the internal list of tables, and due to the 255 character limit of DNS TXT resource records, it cannot be used to store an IOR in excess of this length (In practice, omniORB does not generate IORs in excess of 255 characters, but several of the other ORBs that we tested did). In such cases, the client can make a direct socket connection to the HBD. The client requests the IOR for a given service (in which case the "best" IOR is returned using the same algorithm as the DNS lookup method) or for a specific service and host combination.

Figure 6 illustrates the function of the Service Heartbeat Daemon.

## Adding load balancing and redundancy

The DNS zone in figure 6 uses zeros for the preference, weight, and port values in the SRV record. As mentioned previously, the port for a CORBA server is ephemeral, making it of little use to us. However, the preference and weighting values can be very useful.

When a server registers itself with the HBD, it can specify a priority and weighting. Priorities in SRV records are used just like the priority field of a MX record – first you sort by priority lowest to highest and then you select from all servers with the same priority value. The weight value is used in the SRV record algorithm and allows a server to ask for a smaller or larger proportion of the network load. We select a weight value for each server based upon its capacity, allowing us to take full advantage of each server as we roll in more powerful machines.

To provide a common interface for determining if a server is up or down, we defined a Control interface that the Dispatch interface inherits from. This interface defines administrative procedures to bring services up, take them down, or query their status. When a server is asked for its status, it may simply return a TRUE value, or it may perform complex checks of the resources that it uses (databases, free disk space, etc.) before deciding if it is really up and ready to handle requests.

For the HBD to detect when a server has crashed, it must regularly check the status of each server in its internal table. If one of these checks fail, it sends a multicast IP alert to the other HBDs on the network. When an HBD receives such an alert, it performs its own check of the service, and if it is still found to be down, the server is removed from the internal table. This eliminates the race condition between one HBD marking a service as down and another HBD multicasting an update saying that the same service is up.

There is a degree of latency between a server crashing and all HeartBeat Daemons on the network removing the server from their internal lists; for this reason CORBA clients must be prepared to soft-fail if the attempt to connect to the server fails. As discussed in [MODZ97] certain commercial ORBs (such as Visigenic's Visibroker) offer the ability to transparently fail-over to another application, but this feature is not available in omniORB.

An interesting approach to transparently providing redundancy is that employed by the Eternal system [NARA97]. Eternal intercepts the IIOP protocol communication between CORBA clients and servers, redi-

recting the communication to a reliable multicast group of servers in such a way that multiple servers are given the opportunity to respond to the request. Eternal intercepts the IIOP stream at the operating system level, and as such can be plugged into any ORB that supports IIOP. Unfortunately, Eternal is still under development at the University of California, Santa Barbara and is not yet available to the public.

## Security

We mentioned that all of the CORBA communication takes place over a private LAN. The decision to isolate CORBA traffic allowed us to deploy Datacom Mk2 without building a security infrastructure on top of CORBA. Although several ORB vendors offer CORBA over SSL solutions, they are proprietary, invalidating one of CORBA's biggest selling points. We instead used an open source IPSEC tunnel program to extend this private network, allowing Datacom to use multiple data centers for performance and redundancy. In the future, we hope to move from running software-based tunnels on our servers to using a firmware solution, such as Cisco's VPN product running on our routers.

## Performance

The success of our conversion was tied to tangible results: the new system had to meet or exceed our requirements for it. Using CORBA increases processor and network overhead and our initial performance tests did not simulate the load that the system would be under in the real world. Throughout our development cycle, a large question mark loomed overhead: might we have done all this work only to run into yet another performance barrier?

Our target was 200 transactions per second on each 600Mhz Pentium III machine in the network. After all the servers were running in the lab, we were able to max out a 400Mhz Pentium II machine at more than 400 transactions per second. The design of the system allows for near-linear performance increases as hosts are brought online, so we expect that this model will serve us for the next several years.

## Looking to the Future: CORBA 3

The benefits that CORBA has provided to our application are numerous, but there is an even bigger bonus yet to come: CORBA 3. This collection of specifications addresses many of the annoyances present in the CORBA 2.3 specification and lays the groundwork for integrating CORBA with other component technologies, like ActiveX and JavaBeans.

The new features in CORBA 3 fall into three categories: Internet Integration, Quality of Service (QoS), and the CORBAcomponent architecture. Internet Integration allows CORBA to communicate easier over the Internet by defining a firewall protocol that allows for client-side callbacks over a single TCP/IP connection.

Internet Integration also defines IIOP over SSL, which will allow ORBs from multiple vendors to talk to each other securely. Another big change is the Interoperable Naming Service which lets a client find an IOR using a new URL syntax, like this:

iioploc://ns.targetnet.com/AdDelivery

While this feature might seem to render the Service Heartbeat Daemon obsolete, it still has a single insertion point for updates. Until a hybrid solution that provides a redundant IOR directory appears, we expect that the HBD will continue to be an essential part of our system.

Also in the new spec are QoS extensions, including asynchronous messaging and queue priorities. Specifications for Minimum, Fault-tolerant and Real-time CORBA will make CORBA a viable solution for many applications that were unable to use CORBA version 2.

The CORBAcomponent architecture is intended to separate CORBA into components, and defines the integration of these components with popular scripting languages. This will allow a developer to freely mix and match CORBA 3 components, ActiveX controls, and Enterprise JavaBeans, choosing the best technology for each part of their problem without becoming mired in interoperability issues. This could be the single most important part of CORBA 3, as it will break the present exclusive development model that CORBA 2 forces developers to use.

The CORBA 3 specification will not be published until mid-to-late 2000. Even then, we are not sure how quickly or how much of CORBA 3 we will use. Having spent the time building CORBA into the heart of the ad delivery system gives us the freedom to migrate parts of the system to CORBA 3 at a comfortable pace. Had we chosen a different architecture, we might have been forced to move the entire system in one piece.

### Concluding Remarks

The conversion of Datacom from CGI to CORBA took us a little more than four months. Making the journey from no CORBA knowledge to full implementation was not without its pitfalls – we cut some corners and

skipped over those topics deemed “too advanced” at the time. Nevertheless, we succeeded in creating a system that exceeds its design requirements several times over and allows for almost limitless expansion.

We could have chosen to outsource a solution, but when faced with financial constrictions, solving the problem in-house is often the only option. With a little creative thinking and some investment in learning new technologies, it is possible to deploy a distributed computing model without a large capital investment.

CORBA does not have to remain in the enterprise-computing arena. Examination of projects like Datacom or industry software like the Dents name server [DENTS] and the GNOME environment [GNOME] proves that CORBA/OpenSource integration is viable today. Whether you use CORBA implicitly by contributing to these projects or explicitly as we have, it is clear that CORBA has a bright future in the OpenSource community. We hope that our experience encourages OpenSource developers to seriously consider CORBA as part of their projects.

---

### References

- [FASTCGI] <http://www.fastcgi.com/>
- [MODZ97] Brent E. Modzelewski and David Cyganski, “Interactive-Group Object-Replication Fault Tolerance for CORBA,” *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems*, Portland, Oregon (June 1997).
- [OMG] <http://www.omg.org/>
- [MICO] <http://diamant.vsb.cs.uni-frankfurt.de/~mico/>
- [ILU] <ftp://parcftp.xerox.com/pub/ilu/ilu.html>
- [ORBACUS] <http://www.ORBacus.com>
- [ORBIT] <http://www.labs.redhat.com/orbit/>
- [OMNIORB] <http://www.orl.co.uk/omniORB/omniORB.html>

- 
- [SCHM96] Douglas C. Schmidt, Tim Harrison and Ehab Al-Shaer, "Object-Oriented Components for High-speed Network Programming", *Proceedings of the USENIX Conference on Object-Oriented Technologies*, Monterey, California (June 1995).
- [GNQS] <http://www.gnqs.org/>
- [JAVABEAN] <http://java.sun.com/beans/faq/faq.enterprise.html>
- [RFC2714] V. Ryan, R. Lee, and S. Seligman, *Schema for Representing CORBA Object References in an LDAP Directory*, RFC2714 (October 1999), <ftp://ftp.isi.edu/in-notes/rfc2714.txt>.
- [RFC2052] A. Gulbrandsen and P. Vixie, *A DNS RR for specifying the location of services (DNS SRV)*, RFC2052 (October 1996), <ftp://ftp.isi.edu/in-notes/rfc2052.txt>
- [NARA97] P. Narasimhan, L.E. Moser and P.M. Melliar-Smith, "The Interception Approach to Reliable Distributed CORBA Objects", *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems*, Portland, Oregon (June 1997).
- [DENTS] <http://www.dents.org/>
- [GNOME] <http://www.gnome.org/>



# Outwit: Unix Tool-based Programming Meets the Windows World

Diomidis D. Spinellis

*Department of Information and Communication Systems*

*University of the Aegean*

*GR-83200 Karlovassi, Greece*

dspin@aegean.gr

## Abstract

The ubiquity of Windows-based desktop environments has not been matched by a corresponding emergence of tools supporting the Unix tool composition paradigm. *Outwit* is a suite of tools based on the Unix tool design principles allowing the processing of Windows application data with sophisticated data manipulation pipelines. The *outwit* tools offer access to the Windows clipboard, the registry, relational databases, document properties, and shell links. We demonstrate a number of applications of the *outwit* tools used in conjunction with existing Unix commands, and discuss future directions of our work.

*"GUIs normally make it simple to accomplish simple actions and impossible to accomplish complex actions." — Doug Gwyn*

## 1 Introduction

The remarkably productive environment based around the Unix tool composition philosophy [KP84] is increasingly made irrelevant by the ubiquity of Windows-based desktop environments. In those environments huge, monolithic, "user-friendly", GUI-based applications, binary file formats and databases, and an API of byzantine complexity render useless a large proportion of the Unix toolbox and associated data manipulation techniques. Often, even the blunt application of a tool like *strings* on a Windows binary file such as a Word document can, as part of a small pipeline, produce results that can not be otherwise obtained from the corresponding GUI-based applications.

The increasing sophistication of Windows-based compilers, libraries, and operating system functionality in conjunction with the availability of open-source Unix tool implementations have led to a number of ports of a large portion of the Unix toolchest to the Windows environment. Although such ports provide a rich set of tools, they often fail to address tool integration with the rest of the environment.

Over the last few years we have nurtured and sharpened *outwit*: a small suite of tools, based on the Unix

programming philosophy, yet tightly integrated with the Windows environment. The tools are designed to be seamlessly used with traditional parts of the Unix toolchest as parts of shell-based pipelines. We use these tools on an everyday basis both casually to enhance our system's usability, and as building blocks for more sophisticated applications.

The aim of this paper is to demonstrate that GUI-based platforms, and Windows in particular, are not incompatible with the Unix tool-based approach to prototyping, reuse, shell programming, and implementation based on incremental refinement. In the following sections we will outline the design principles of our approach, describe the tools in our collection, comment on their implementation effort, illustrate some typical uses, and discuss our work in the context of other efforts and future directions.

## 2 Design Rationale and Principles

Typical GUI-based environments and applications provide a fixed set of features wrapped in a standardized user-friendly interaction paradigm based on windows, icons, menus, and pointing (WIMP). Although this approach enables inexperienced users to quickly master applications and produce results, its limits rapidly become apparent when one tries to perform a task not foreseen by the original developers of the application. In many cases it is difficult to combine different applications together to achieve a complex task, repetitive tasks

can not be automated, and interaction sequences can not be manipulated for future reuse. In short, GUI based applications are inflexible factories rather than tools; end-users must accept the end products or wait for central planning to devise a new ones.

Some applications, to overcome these problems, provide an object model and a script-based programming interface. Such interfaces, however, still carry with them the weight of the application, do not provide a clean way to combine different applications, and, in our experience, are typically fragile, and difficult to use in a non-interactive setting.

An alternative approach to productivity in an interactive environment was popularized by the Unix tool-based and shell programming approach [KP84]. Unix tools can be easily combined to form powerful data manipulation sequences. Where such sequences are to be reused or repeated Unix shell-based programming can be used to package and automate them. The advantages of the Unix-based approach to interaction prompted us to examine the possibility of offering these benefits within the context of the ubiquitous Windows environment. A large part of the Unix toolchest has already been ported to Windows as part of open-source and commercial efforts. However, the tools provided typically function isolated from the GUI environment within a text-based command shell. Data interchange with the GUI applications is often limited to the use of textual files which are in any case rare as most GUI applications use undocumented binary file formats. We therefore focused our attention on providing mechanisms for integrating the capabilities provided by the Unix toolchest with the data sources and sinks available in typical GUI applications.

Our tools were designed along the principles used by most Unix-based tools:

- do a single job well,
- avoid decorative headers and trailing information,
- accept input created by other tools,
- generate output that can be used by other tools, and
- be capable of stand-alone execution without user intervention.

It is interesting and instructive to contrast tools based on these principles against the behemoths that populate GUI-based office productivity suites.

We aimed our tools to areas where data amenable to Unix-style pipeline processing could be extracted from difficult-to-access Windows sources. All the tools are console-based native Win32 applications using their standard input and output as the data source and sink. They can be used both from the standard Windows shells (*command* and *cmd32*) and from Unix-derived shells like *bash* and *ksh* [Kor94].

Due to the complexity of the Windows API the external tool simplicity is often inversely proportional to the code required to implement the tool. We found however, that the effort required to implement such tools was often amortized by less mouse clicks and other repetitive GUI operations within a matter of days.

### 3 Tool Descriptions

The tools we developed aim to make data that is typically accessed through GUI-based applications usable via standard command pipelines. We therefore aimed at providing integration mechanisms for the Windows clipboard, relational databases, the Windows registry, OLE document properties, and shell-namespace links. In the following paragraphs we describe the facilities offered by each tool.

#### 3.1 Clipboard Integration

Most Windows GUI applications and a number of the native system controls (widgets) allow the user to copy and paste data to and from a global clipboard area. This user-driven interprocess communication mechanism is used for sharing data between applications. The clipboard data can be in an application-native format, in which case it can be processed only by the application family that is aware of that format, or it can be in a number of documented public formats. The Windows system will automatically convert between a number of compatible formats such as different types of bitmaps.

Our *winclip* tool provides shell-based clipboard access. When run with a *-c* (copy) argument it will copy the data it reads from its standard input to the Windows clipboard:

```
ls -l | winclip -c
```

Currently all data is copied to the clipboard as text. When *winclip* is run with a *-p* (paste) argument it will

paste the data of the Windows clipboard to its standard output:

```
winclip -p | wc -w
```

Currently *winclip* supports the following clipboard data formats for its output:

- text,
- bitmap — output in *ppm* [P<sup>+</sup>93] format, and
- drag file list — files “copied” using the Windows Explorer interface are output as lines of the respective file paths.

### 3.2 Database Access

The Windows platform is increasingly used as a client for database systems. Database communication is often performed using the Open Database Connectivity (ODBC) application programming interface [Mic97]. ODBC is based on the Call-Level Interface (CLI) specifications from X/Open and ISO/IEC [ISO95] for database APIs and uses the Structured Query Language (SQL) as its database access language. An ODBC driver manager, provided as part of the Windows platform, allows the installation of the drivers supplied by most database vendors. Using an appropriate ODBC driver Windows-based applications can transparently communicate with a wide variety of database engines. Although most database vendors supply text-based SQL database access tools, these are typically database specific and difficult to use for shell-based programming.

Capitalizing on the strengths of the ODBC interface we wrote the *odbc* tool which prints the results of an SQL SELECT command run on any database for which an appropriate data source has been defined:

```
mail 'odbc uDB 'select email from users''
```

Two arguments must always be specified as part of the tool invocation: the name of the data source driver (the database communication configured in the local machine), and a select statement. In addition, one can specify the record and field separator that will delimit the output results, and a user identifier and an authorization string pair that will be used for logging into the database. All data types are currently output as text according to the default driver-supplied data conversions.

### 3.3 Registry Operations

The Windows registry is a database that stores system management information in a hierarchically structured tree. Each node in the tree, called a key, can contain both subkeys and data entries. A data entry can contain a text string, an integer, or binary data. Windows systems use the registry for storing all configuration data including data related to the operating system, computer hardware, applications, and user preferences. The registry is physically divided into a part containing configuration information about the machine and the operating system, and a part containing user-specific information. The user-specific part is often replaced during the login procedure to allow user roaming or the sharing of a machine among different users. In addition, a special category of keys, called dynamic keys, are updated at runtime by device drivers to monitor quantities such as data transfer rates, processor utilization, and dropped packets. The registry can be manipulated through the GUI registry editor *regedit*. Although this program supports a text import/export format and an, almost undocumented, command-line interface, the textual representation used is not amenable to Unix-tool based processing and the command-line interface is too rigid for many useful applications. Specifically, the text format used by *regedit* places the key names in a separate line followed by a number of key values. In addition, the command-line invocation of the tool executes only in the background and does not allow the text-based representation to be redirected as input or output.

In order to overcome these difficulties we designed and implemented *winreg*, a text-based tool that can be used to read and modify registry data. Each text line represents a single key. Three fields, separated by a user configurable field separator, contain the key name, its type, and its value. The tool can be invoked with the name of a part of the registry as its argument to print the textual representation of the registry tree from that point downwards on its standard output, or it can read the textual representation of some registry keys from its standard input and enter them into the registry. In addition, one can specify whether the output of *winreg* shall include the key names, types, or values. As an example:

```
winreg -nt HKEY_LOCAL_MACHINE\System\
  \CurrentControlSet\Control\ComputerName\
  \ComputerName
```

will display a machine's name.

### 3.4 Document Properties

A number of applications in the Windows environment expose document meta information such as a document's title, author, keywords, and number of pages it contains by using a standard document information property set. This set is part of Microsoft's Component Object Model (COM) structured storage facilities. Three property sets are currently defined:

**the summary information** containing the document's title, subject, author, keywords, comments, revision, editing time, the time the document was created, printed, and saved, the number of pages, words, and characters, and the name of the creating application,

**the document summary information** containing the document's category, presentation target, number of paragraphs, lines, notes, slides, and the names of the company and the project's manager, and

**the user-defined property set** where users can create and store named properties to store additional document information.

Document properties are typically accessed from within applications, or through a file context menu of the Windows GUI shell.

Our *docprop* tool provides Unix shell programmable access to these properties. The program takes as arguments an optional output format specification string and a list of filenames. The format specification string can contain arbitrary text, system or user-defined property names enclosed in braces, and the usual C language backslash escapes. When executed the program will iterate over the filename list and, for each file, will output the format string replacing the property names enclosed in braces with their respective values. In addition, *docprop* provides an internally synthesized property name called *Filename* to allow the printing of each filename. The following example will print the document name and author name of all Word documents in a directory:

```
docprop -f '{Filename}\t{Author}\n' *.doc
```

### 3.5 Shell Links

Shell links, also known as shortcuts, are data objects that contain information used to access another object in the GUI shell's namespace. They are superficially similar to

the Unix symbolic links; their most important difference is that the file or object they point to is *not* resolved automatically by the kernel when an application accesses a link. In addition, if the target object is moved, the system will attempt to locate the target object in its new position when the link needs to be resolved. Links typically operate only at the level of the GUI shell. The types of objects that can be accessed through shell links include files, folders, disk drives, and printers.

Despite our philosophical objections to the shell link concept we found the need for a tool to resolve such links at a textual level. We thus discovered that the operation that is trivially performed on a Unix system by the *readlink(2)* system call needed 60 lines of C code and 11 calls to the Win32 API in the Windows environment. Our resultant *readlink* tool accepts as its single argument the name of a shell link and outputs the name of the link's target. Dynamic resolving of targets on the move works as advertised:

```
$ readlink s.lnk
C:\src\win32lib\port.c

$ move port.c foo
C:\src\win32lib\port.c =>
  C:\src\win32lib\foo\port.c [OK]

$ readlink s.lnk
C:\src\win32lib\foo\port.c
```

## 4 Implementation Notes

The realization of all tools is based on the possibility to call Win32 API functions from within so-called *console*, i.e. text-based, applications. Although all examples on the use of the Win32 API functions provided with the Microsoft documentation are complete GUI-based applications, in practice most API functions can be called without a problem from text-based programs. This technique forms the basis for integrating the text-based and the GUI worlds. All our programs, implemented in C and C++, call the appropriate Win32 API functions to transfer data to and from the GUI world and use *stdio* I/O functions to interface with the Unix shell world.

At 1440 lines of code the total implementation effort for the *outwit* suite is embarrassingly modest given the capabilities it provides. The code size of each tool is detailed in Table 1. Most tools are at the same time more versatile, powerful, and smaller than the corresponding GUI-based toy examples demonstrating similar capabilities. With increasing use of the *outwit* suite we expect to add more tools and enhance the capabilities of the exist-



Tool name	L.O.C.
readlink	141
odbc	151
docprop	408
winreg	545
winclip	195
Total	1440

Table 1: Tool code size metrics

ing ones. However, given that the power of our approach relies on the synergies of tool composition and not on the features of a single tool, we expect both the number of tools and the size of each tool to remain fairly small.

The greatest hurdle in the development of each tool was the Win32 API. Many API functions are incompletely or inconsistently documented [Spi98] while the sheer size of the API is formidable: it currently comprises 149 different data types, 2193 basic functions, and 1499 error codes. We were also troubled by subtle incompatibilities between operating system versions that could create portability problems, the lack of documentation on the possible function error return values, and the overly complicated interface provided by some functions. We were however pleasantly surprised by the quality of the ODBC interface which stands apart in documentation quality from the rest of the Windows API.

Many API functions are based on special *handles* which must be carefully allocated and deallocated. We believe that this must be a source of reliability problems in the large GUI-based applications. In contrast, the *outwit* tools will simply process data and exit, thereby sidestepping various resource leak problems.

In order to maximize the applicability of the *outwit* tools, all of them are written as native Win32 applications and do not rely on an intermediate Unix porting layer. They can thus be used together with any collection of Unix tools that allows the execution of Win32 programs. All tools can be used under Windows 95, 98, 2000, and NT.

## 5 Exemplar Uses

We believe that the mode of work enabled and demonstrated by the *outwit* tool suite is more important than the tools themselves. In the following paragraphs we therefore present some — motivating we hope — examples of how Unix tool-based programming can be applied to the Windows GUI world.

### 5.1 Winreg

Tool-based access to the Windows registry can allow, in combination with other Unix tools, sophisticated registry manipulations that are impossible within the GUI-based registry editor. A commonly used idiom involves processing Windows registry data as the output of *winreg* using Unix tools like *sed* and *awk* and redirecting their output back to the registry via *winreg*. The following example will change all user registry references from `c:/home` to `d:/home`:

```
winreg HKEY_CURRENT_USER |
sed -n 's/C:\\home/D:\\home/gp' |
winreg
```

*Winreg* is also often used to extract system information from the registry. The name of the currently logged-in user can be stored in a shell variable using the following construct:

```
LOGIN='winreg \
HKEY_LOCAL_MACHINE\Network\Logon |
awk '/username/{print $3}' '
```

In addition, network settings can be obtained from the registry keys `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\VxD\MSTCP` and `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\VxD\DHCP`.

One other application of *winreg* involves accessing system performance information similar in nature to the data provided by the Unix *vmstat* command. The following command sequence will save system performance information (CPU load, data I/O, cache and memory system data in the form of raw counter values) to a file for later statistical processing. This functionality is not provided by the system-supplied performance monitors and would otherwise require the implementation of a separate program:

```
while :
do
winreg -dt HKEY_DYN_DATA\PerfStats\StatData
sleep 5
done >/var/log/perfdata
```

### 5.2 Winclip

Winclip can often be used as a quick way to provide functionality lacking from GUI applications. The command `winclip -c </home/signature`, when tied to a

keyboard shortcut, can be used to quickly obtain a copy of the user's signature file contents within any application. Similarly, the following sequence can be used to trivially encrypt or decrypt the text contained in the clipboard using the *rot13* substitution cypher:

```
winclip -p |
tr ' [a-z] [A-Z] ' ' [n-z] [a-m] [N-Z] [A-M] ' |
winclip -c
```

(This "encryption" method is sometimes used in certain newsgroups to discourage casual reading of joke punch lines or offensive jokes.)

By using the point and click metaphor, GUI interfaces to the filesystem such as the Windows Explorer often provide a swifter method for selecting files than typing file names at the command prompt. *Winclip* allows the integration of the two interaction approaches. The following sequence will open up a console window with its current directory set to the directory copied to the clipboard from the Windows Explorer:

```
\bin\echo -n "start command /k cd " >$$$.bat
winclip -p >>$$$.bat
start $$$$.bat
```

The Windows Explorer is also our method of choice for visually selecting groups of files. Having selected a group of files they can then be copied to the clipboard (as drop targets) and processed using text-based commands:

```
sed '/^$/d' 'winclip -p' | wc -l
```

*Winclip*, in conjunction with a pipeline-based image manipulation package such as *netpbm* [P<sup>+</sup>93], can also be used to automate the processing of graphics images that have been copied to the clipboard. When processing a large number of images this method of work can save the user from the tiresome repetitive operations that would have to be performed on a GUI-based graphics manipulation package. Alternatively, *winclip* can be used to add new facilities to a GUI package by processing the image in the clipboard. As an example, the following sequence of commands will crop, scale, quantize, and convert the clipboard bitmap into a GIF file:

```
winclip -p |
pnmcrop |
pnmscale 0.5 |
ppmquant 256 |
ppmtogif >file.gif
```

### 5.3 Odbc

SQL and a number of Unix tools offer complementary approaches for obtaining the same result. SQL can be used to select records and fields, group records, join tables, and sort in the same way as the Unix *grep*, *awk*, *uniq*, *join* and *sort* commands. There are however cases where applying Unix tools to data obtained from SQL databases can enhance productivity. One example concerns the combination of different databases (hosted on different servers and applications). Using our *odbc* tool the following sequence selects, merges, sorts, and prints fields from two different databases:

```
(
odbc ACorpDB "select FullName, Phone from
Employees"
odbc BCorpDB "select surname, name,
phonenum from personnel"
) | sort | pr
```

*Odbc* is also useful when integrating Unix and Windows-based systems. A password file can be extracted from a relational database and copied to a Unix file using the following command:

```
odbc -F: userDB "select * from passwd" \
>/nfs/host/passwd
```

(The security implications of such functionality are left as an exercise to the reader.)

Finally, given the existence of text-based graph creation tools like *GNUplot* combining *odbc*, Unix text manipulation tools, and *GNUplot* can be an efficient way to automate the generation of graphs from databases.

### 5.4 Readlink and docprop

*Readlink* and *docprop* are often used together to process GUI file data. *Readlink* is typically used to resolve shell-namespace symbolic links in a sequence similar to the following:

```
case $FNAME in
*.lnk) FNAME='readlink $1' ;;
esac
```

*Docprop* can be used to create and consolidate indices of document titles and authors. *Docprop* can also be used

PROGRAMME		Form INF3	
WORKPACKAGE DESCRIPTION			
For WORKPACKAGE N° 0			
WP title:	Project Management		
WP leader:	P0		
WP contributors:	All		
Start month:	1	End month:	24
Estimated labour effort in person-months: 12			
Activities related to this workpackage will address the following items: [...]			
Constituent task(s):		Task leader:	
T0.1	Definition of technical & operational internal procedures and Quality aspects of the Project	P0	
T0.2	Contractual Reporting	P0	
T0.3	Organisation of Kick-off, Interim, Concertation, and Final meetings	Hosting partner	
Deliverables (max 3 per WP)		Month Due:	
D0.1	Project Quality Plan	1	
D0.#	Progress reports and cost statements	6,12,18,24	
D0.6	Final report	24	

Figure 1: Workpackage description form.

to gather statistical data from document files. The following sequence will print a list of application names that were used to edit the most recently used files, sorted in order of application popularity.

```
for f in /windows/recent/*.lnk
do
    readlink $f
done |
xargs docprop -f '{Application}\n' |
sort |
uniq -c |
sort -n
```

## 5.5 Document Processing

The integration of the GUI and Unix tool worlds can be a particularly productive option when processing textual data using (or having to use) a GUI-based word processor. In the following paragraphs we will present an exemplar case concerning processing textual data for a funding proposal. The work to be funded was divided into eight workpackages; each workpackage had to be described using a form similar to the one appearing in Figure 1.

In addition to a description for each workpackage, the funding body required the provision of a list of deliverables ordered by delivery month, and a GANTT chart for the whole project. We also wanted a current list of tasks in order to distribute work between the project participants. All these items are different representations of the data included in the workpackage descriptions. The combined use of *winclip* and some Unix tools allowed us

PROGRAMME

Form INF4

DELIVERABLE DESCRIPTION

Due end of month n°	WP n°	Deliverable Title	Deliverable Type and Status
	0	D0.1 Project Quality Plan	Report, Restricted
12,18,24	0	D0.# Progress reports and cost statements	Report, Confidential
	0	D0.6 Final report	Report, Restricted
	1	D1.1 Requirement Specification	Report, Confidential
	1	D1.2 Architecture Design	Report, Confidential
	2	D2.1 Draft Specification of interlingua	Report, Confidential
	2	D2.2 Greek and English compatible revised interlingua	Report, Confidential
	2	D2.3 Interlingua for implementation	Report, Confidential
	3	D3.1 Metalevel Specification	Report, Confidential
	3	D3.2 Interlingua for implementation	Report, Confidential
	4	D4.1 Report on selection of resources and their use in the system	Report, Restricted
	4	D4.2 Prototype Resources	Data, Confidential

Figure 2: Project deliverable list based on *awk* data.

to automate the creation of these items. All sequences we used involved the massaging of the clipboard data (which contained the workpackage descriptions) using *awk* into a format appropriate to be pasted back into another table or application.

The following script was used to create a list of tasks:

```
winclip -p |
awk "-F\t" '/^T[0-9]/{print $1 " " $2}' |
winclip -c
```

The script below was used to create a list of deliverables to be pasted back into a summary form in the original application:

```
winclip -p |
awk "-F\t" '/^D[0-9]/ {
    print $3 "\t" substr($1, 2, 1) \
        "\t" $1 "\t" $2
}' |
winclip -c
```

The corresponding table is depicted in Figure 2.

Finally, the following script was used to create a workpackage list together with start dates and durations in a format suitable for pasting into Microsoft Project:

```
winclip -p | awk '
BEGIN {FS = "\t" }
/For WORK/ {split($0, a, "\t")}
```

```

/^WP title/ {WP = $2}
/^Start month/ {
    print "WP" a[4] "\t" \
        WP "\t" \
        ($4 - $2 + 1) * 31 "ed\t" \
        "1/" ($2 - 1) % 12 + 1 "/" 2000 +
        int(($2 - 1)/12)
}' |
winclip -c

```

The output was of the following form:

```

WP0 Project Management 744ed 1/1/2000
WP1 Requirements Analysis 124ed 1/1/2000
WP3 Metalevel Specification 279ed 1/4/2000
...

```

The resulting GANTT chart is depicted in Figure 3.

## 6 Related Work

A number of efforts have been undertaken to provide the functionality of the Unix tools in the Windows environment. It is interesting that only one of them is based on the POSIX subsystem provided under Windows NT as an alternative to Win32. The main drawback of the POSIX subsystem is that processes running in it are essentially isolated from the rest of the system. Most porting efforts complement our *outwit* suite by providing the necessary tools needed to utilize our offerings. In some cases facilities for cooperating with GUI applications are also provided.

Our *wux* port of Unix tools to the 16 bit Windows environment [Spi94] demonstrated the possibility of implementing true multitasked processing of pipeline command sequences under the Windows environment, but did not offer any additional integration facilities.

The UWIN port [Kor97] of the Unix tools and libraries supports all X/Open Release 4 headers, interfaces, and commands. It supports the shell namespace links (shortcuts) by mapping them internally to Unix symbolic links. It is interesting to note that UWIN resolves links by reverse engineering the OLE link file format rather than calling the corresponding Win32 API functions. UWIN provides access to the clipboard via the `/dev/clipboard` device and to the Windows registry via a virtual file system that is mounted at `/reg`. Both facilities are however only available to programs that have been compiled under the UWIN environment. As a result, native operating system console commands (e.g. *dir*) and commands that have been compiled without using the UWIN libraries can not directly access the clipboard and the registry. On the contrary, *outwit* tools provide clipboard and registry

access to all character-based console programs, including 16-bit legacy applications.

*Cygwin* [Noe98] is a full Win32 porting layer for Unix applications. It supports the GNU development tools and allows the effortless port of many Unix programs by supporting almost all POSIX.1/90 calls and other Unix version-specific functionality. A novel integration aspect of *cygwin* is the provision of the `/dev/windows` pseudo-device which can be used as a source for Windows messages (user input from the keyboard, the mouse, and IPC events).

Finally, *OpenNt* [Wal97] is a complete porting and runtime environment that can be used to migrate application source, developed on traditional Unix systems, directly to Windows NT. Ported software includes many X11R5 clients and over 200 other utilities. It is implemented using an enhanced Windows NT POSIX subsystem. As the POSIX subsystem is isolated from the Win32 subsystem, integration between the two worlds is offered through the filesystem, the desktop, a special function to execute Win32 applications, and socket-based IPC.

A number of technologies support high-level programming in the Windows environment. These include languages such as Visual Basic [Boc99], Perl [WCSP96], and TCL/TK [Ous94], and integration mechanisms such as OLE automation and Windows scripting. These approaches, based on a programming language, are superior for programming in the large and developing applications in a top down manner. Tool-based approaches such as the one we advocate complement such an environment offering a different development path. As tools are directly used from the shell command line in a casual user interaction pattern repetitive tasks are gradually automated as shell scripts and subsequently, as they mature, packaged as applications. We believe that this bottom-up evolutionary style of development should be available together with other approaches.

## 7 Conclusions and Further Work

The tools we described can be extended in breadth and depth. We are currently planning to enhance *winclip* with additional data types to be able to handle sound data as input and output, as well as bitmap data as input. *Odbc* can also be improved by adding the ability to modify database data using the SQL *update* command. Such a facility will be useful for transferring data between databases through a pipeline with two *odbc* commands at its ends. In addition, with the advent of Windows 2000 which supports the *IFilter* interface, *docprop*



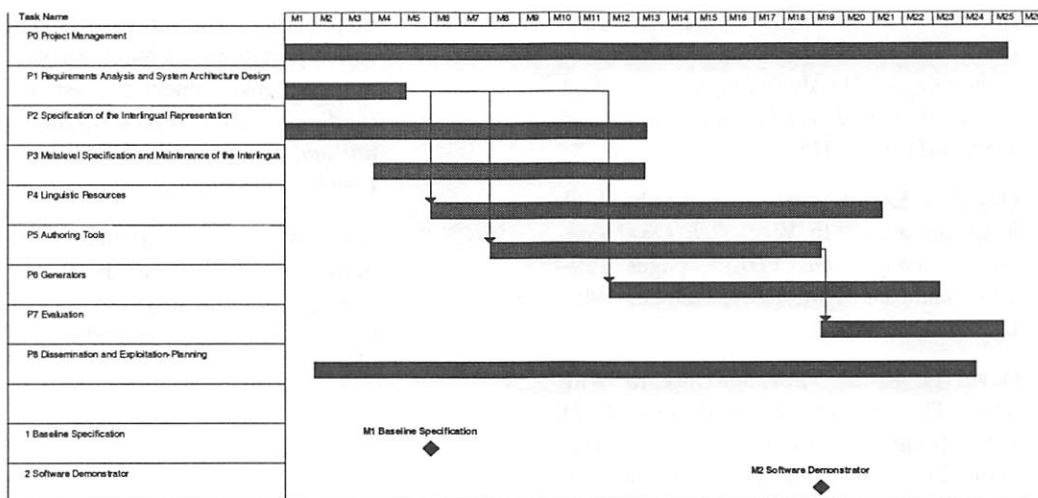


Figure 3: Project GANTT chart based on *awk* data.

can be extended to retrieve a textual representation of any object that supports the respective interface. Finally, the migration of many operating system databases to the Windows 2000 Active Directory has created the need for a new text-based tool to manipulate them.

Many Windows applications expose an object model of the data and operations they support. Environments such as Visual Basic and language extensions such as the Perl OLE module allow programmers to access and manipulate Windows applications and their data from external programs. Although we feel that shell-based programming can not compete against environments tailored to support this mode of programming, a simple text-based command to access OLE automation objects could prove a valuable addition to our *outwit* suite. On the same front, but from a different angle, we are examining how Unix tools can be repackaged as OLE components to be used within visual programming environments [Spi99].

One final dimension of work that will increase the applicability of *outwit* concerns internationalization. All Win32 API functions dealing with strings are provided in 8-bit character and Unicode (wide character) versions. Although low-end Windows systems such as Windows 95/98 are not supporting Unicode for most of the API functions, Windows NT and Windows 2000 provide full Unicode support. As a result, the clipboard, filenames, registry keys, and databases may contain the full repertoire of Unicode characters. Currently *outwit* is coded and compiled as an 8-bit character application. Following the lead established by the Plan 9 internationalization efforts [PT93] a reasonable approach would be to adapt all *outwit* tools to use a multibyte (e.g. UTF-8)

representation for their textual input and output. This approach would work with many existing Unix tools; we expect programs that explicitly deal with characters and will need to be modified (e.g. *grep*, *sort*, *sed*, and *tr*) to follow a similar approach.

In the previous paragraphs we demonstrated that the rise of GUI-based environments does not mean that tool building and Unix shell programming are less relevant today than they were 25 years ago. The lack of support for tool-based programming in the Windows environment is a result of market dynamics rather than an inherent limitation of the environment. Integrated GUI-based systems are probably the only applications that can be marketed and sold at a profit. However, modest effort invested in tool building will produce tools that can be used as building blocks in concert with the large number of existing Unix tools and the powerful shells.

## Availability

The tools described are available online at:

<http://softlab.icsd.aegean.gr/~dspin/sw/outwit>

## References

- [Boc99] David Bector. *Microsoft Office 2000 Visual Basic Fundamentals*. Microsoft Press, Redmond, WA, USA, 1999.

- [ISO95] International Organization for Standardization, Geneva, Switzerland. *Information technology — Database languages — SQL — Part 3: Call-Level Interface (SQL/CLI)*, 1995. ISO/IEC 9075-3:1995.
- [Kor94] David G. Korn. Ksh - an extensible high level language. In *Very High Level Languages Symposium (VHLL)*, pages 129–146, Santa Fe, NM, USA, October 1994. Usenix Association.
- [Kor97] David G. Korn. Porting Unix to Windows NT. In *Proceedings of the USENIX 1997 Annual Technical Conference*, Anaheim, CA, USA, January 1997. Usenix Association.
- [KP84] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice-Hall, 1984.
- [Mic97] Microsoft Corporation. *Microsoft ODBC 3.0 Programmer's Reference and SDK Guide*. Microsoft Press, Redmond, WA, USA, 1997.
- [Noe98] Geoffrey J. Noer. Cygwin32: A free Win32 porting layer for UNIX applications. In *Proceedings of the 2nd USENIX Windows NT Symposium*, Seattle, WA, USA, August 1998. Usenix Association.
- [Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.
- [P<sup>+</sup>93] Jef Poskanzer et al. NETPBM: Extended portable bitmap toolkit. Available online <ftp://ftp.x.org/contrib/utilities/>, December 1993. Release 7.
- [PT93] Rob Pike and Ken Thompson. Hello world. In *USENIX Technical Conference Proceedings*, pages 43–50, San Diego, CA, USA, Winter 1993. Usenix Association.
- [Spi94] Diomidis Spinellis. Wux: Unix tools under Windows. In *USENIX Conference Proceedings*, pages 325–336, San Francisco, CA, USA, Winter 1994. Usenix Association.
- [Spi98] Diomidis Spinellis. A critique of the Windows application programming interface. *Computer Standards & Interfaces*, 20:1–8, November 1998.
- [Spi99] Diomidis Spinellis. Explore, excogitate, exploit: Component mining. *IEEE Computer*, 32(9):114–116, September 1999.
- [Wal97] Stephen R. Walli. OPENNT: UNIX application portability to Windows NT via an alternative environment subsystem. In *Proceedings of the USENIX Windows NT Symposium*, Seattle, WA, USA, August 1997. Usenix Association.
- [WCSP96] Larry Wall, Tom Christiansen, Randal L. Schwartz, and Stephen Potter. *Programming Perl*. O'Reilly and Associates, Sebastopol, CA, USA, second edition, 1996.

# Plumbing and Other Utilities

Rob Pike

*Bell Laboratories*  
*Murray Hill, New Jersey 07974*  
rob@plan9.bell-labs.com

**Abstract:** *Plumbing is a new mechanism for inter-process communication in Plan 9, specifically the passing of messages between interactive programs as part of the user interface. Although plumbing shares some properties with familiar notions such as cut and paste, it offers a more general data exchange mechanism without imposing a particular user interface.*

*The core of the plumbing system is a program called the plumber, which handles all messages and dispatches and reformats them according to configuration rules written in a special-purpose language. This approach allows the contents and context of a piece of data to define how it is handled. Unlike with drag and drop or cut and paste, the user doesn't need to deliver the data; the contents of a plumbing message, as interpreted by the plumbing rules, determine its destination.*

*The plumber has an unusual architecture: it is a language-driven file server. This design has distinct advantages. It makes plumbing easy to add to an existing, Unix-like command environment; it guarantees uniform handling of inter-application messages; it off-loads from those applications most of the work of extracting and dispatching messages; and it works transparently across a network.*

## 1. Introduction

Data moves from program to program in myriad ways. Command-line arguments, shell pipe lines, cut and paste, drag and drop, and other user interface techniques all provide some form of interprocess communication. Then there are tricks associated with special domains, such as HTML hyperlinks or the heuristics mail readers use to highlight URLs embedded in mail messages. Some systems provide implicit ways to automate the attachment of program to data—the best known examples are probably the resource forks in MacOS and the file name extension ‘associations’ in Microsoft Windows—but in practice humans must too often carry their data from program to program.

Why should a human do the work? Usually there is one obvious thing to do with a piece of data, and the data

itself suggests what this is. Resource forks and associations speak to this issue directly, but statically and narrowly and with little opportunity to control the behavior. Mechanisms with more generality, such as cut and paste or drag and drop, demand too much manipulation by the user and are (therefore) too error-prone.

We want a system that, given a piece of data, hands it to the appropriate application by default with little or no human intervention, while still permitting the user to override the defaults if desired.

The plumbing system is an attempt to address some of these issues in a single, coherent, central way. It provides a mechanism for formatting and sending arbitrary messages between applications, typically interactive programs such as text editors, web browsers, and the window system, under the control of a central message-handling server called the *plumber*. Interactive programs provide application-specific connections to the plumber, triggering with minimal user action the transfer of data or control to other programs. The result is similar to a hypertext system in which all the links are implicit, extracted automatically by examining the data and the user's actions. It obviates cut and paste and other such hand-driven interprocess communication mechanisms. Plumbing delivers the goods to the right place automatically.

## 2. Overview

The plumber is implemented as a Plan 9 file server [Pike93]; programs send messages by writing them to the plumber's file `/mnt/plumb/send`, and receive messages by reading them from *ports*, which are other plumber files in `/mnt/plumb`. For example, `/mnt/plumb/edit` is by convention the file from which a text editor reads messages requesting it to open and display a file for editing. (See Figure 1.)

The plumber takes messages from the `send` file and interprets their contents using rules defined by a special-purpose pattern-action language. The language specifies any rewriting of the message that is to be done by the plumber and defines how to dispose of a mes-

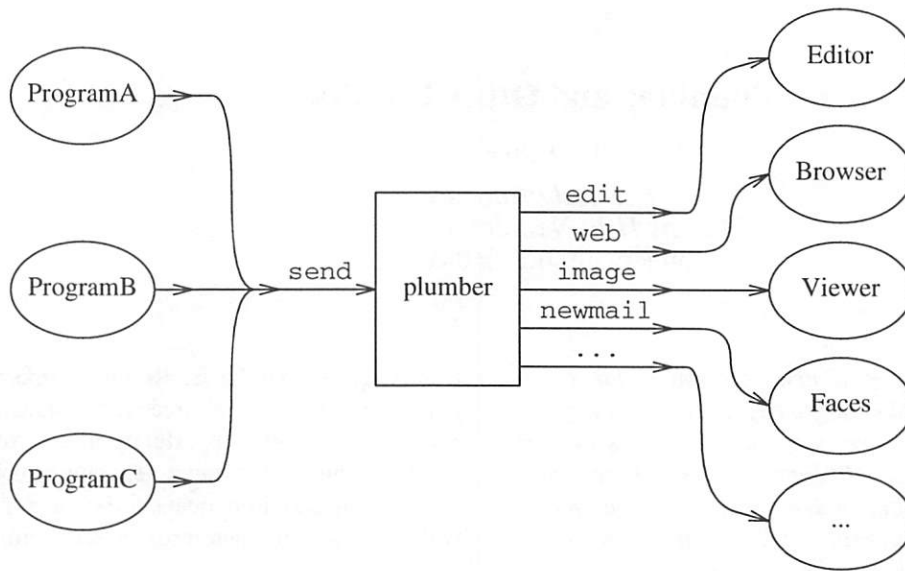


Figure 1. The plumber controls the flow of messages between applications. Programs write to the file `send` and receive on 'ports' of various names representing services such as `edit` or `web`. Although the figure doesn't illustrate it, some programs may both send and receive messages, and some ports are read by multiple applications.

sage, such as by sending it to a port or starting a new process to handle it.

The behavior is best described by example. Imagine that the user has, in a terminal emulator window, just run a compilation that has failed:

```
% make
cc -c rmstar.c
rmstar.c:32: syntax error
...
```

The user points the typing cursor somewhere in the string `rmstar.c:32:` and executes the `plumb` menu entry. This causes the terminal emulator to format a plumbing message containing the entire string surrounding the cursor, `rmstar:32:`, and to write it to `/mnt/plumb/send`. The plumber receives this message and compares it sequentially to the various patterns in its configuration. Eventually, it will find one that breaks the string into pieces, `rmstar.c`, a colon, `32`, and the final colon. Other associated patterns verify that `rmstar.c` is a file in the current directory of the program generating the message, and that `32` looks like a line number within it. The plumber rewrites the message, setting the data to the string `rmstar.c` and attaching an indication that `32` is a line number to display. Finally, it sends the resulting message to the `edit` port. The text editor picks up the message, opens

`rmstar.c` (if it's not already open) and highlights line 32, the location of the syntax error.

From the user's point of view, this process is simple: the error message appears, it is 'plumbed', and the editor jumps to the problem.

Of course, there are many different ways to cause compiler messages to pop up the source of an error, but the design of the plumber addresses more general issues than the specific goal of shortening the compile/debug/edit cycle. It facilitates the general exchange of data among programs, interactive or otherwise, throughout the environment, and its architecture—a central, language-driven file server—although unusual, has distinct advantages. It makes plumbing easy to add to an existing, Unix-like command environment; it guarantees uniform handling of inter-application messages; it off-loads from those applications most of the work of extracting and dispatching messages; and it works transparently and effortlessly across a network.

This paper is organized bottom-up, beginning with the format of the messages and proceeding through the plumbing language, the handling of messages, and the interactive user interface. The last sections discuss the implications of the design and compare the plumbing system to other environments that provide similar services.



### 3. Format of messages

Since the language that controls the plumber is defined in terms of the contents of plumbing messages, we begin by describing their layout.

Plumbing messages have a fixed-format textual header followed by a free-format data section. The header consists of six lines of text, in set order, each specifying a property of the message. Any line may be blank except the last, which is the length of the data portion of the message, as a decimal string. The lines are, in order:

The source application, the name of the program generating the message.

The destination port, the name of the port to which the messages should be sent.

The working directory in which the message was generated.

The type of the data, analogous to a MIME type, such as `text` or `image/gif`.

Attributes of the message, given as blank-separated `name=value` pairs. The values may be quoted to protect blanks or quotes; values may not contain newlines.

The length of the data section, in bytes.

Here is a sample message, one that (conventionally) tells the editor to open the file `/usr/rob/src/mem.c` and display line 27 within it:

```
plumbtest
edit
/usr/rob/src
text
addr=27
5
mem.c
```

Because in general it need not be text, the data section of the message has no terminating newline.

A library interface simplifies the processing of messages by translating them to and from a data structure, `Plumbmsg`, defined like this:

```
typedef struct Plumbattr Plumbattr;
typedef struct Plumbmsg Plumbmsg;

struct Plumbmsg
{
    char *src; /* source application */
    char *dst; /* destination port */
    char *wdir; /* working directory */
    char *type; /* type of data */
    Plumbattr *attr; /* attribute list */
    int ndata; /* #bytes of data */
    char *data;
};
```

```
struct Plumbattr
{
    char *name;
    char *value;
    Plumbattr *next;
};
```

The library also includes routines to send a message, receive a message, manipulate the attribute list, and so on.

### 4. The Language

An instance of the plumber runs for each user on each terminal or workstation. It begins by reading its rules from the file `lib/plumbing` in the user's home directory, which in turn may use `include` statements to interpolate macro definitions and rules from standard plumbing rule libraries stored in `/sys/lib/plumb`.

The rules control the processing of messages. They are written in a pattern-action language comprising a sequence of blank-line-separated *rule sets*, each of which contains one or more *patterns* followed by one or more *actions*. Each incoming message is compared against the rule sets in order. If all the patterns within a rule set succeed, one of the associated actions is taken and processing completes.

The syntax of the language is straightforward. Each rule (pattern or action) has three components, separated by white space: an *object*, a *verb*, and optional *arguments*. The object identifies a part of the message, such as the source application (`src`), or the data portion of the message (`data`), or the rule's own arguments (`arg`); or it is the keyword `plumb`, which introduces an action. The verb specifies an operation to perform on the object, such as the word `'is'` to require precise equality between the object and the argument, or `'isdir'` to require that the object be the name of a directory.

For instance, this rule set sends messages containing the names of files ending in `.gif`, `.jpg`, etc. to a program, `page`, to display them; it is analogous to a Windows association rule (here and in some later examples, long patterns have been folded to fit):

```
# image files go to page
type is text
data matches '[a-zA-Z0-9_\-./]+'
data matches '([a-zA-Z0-9_\-./]+)
        \.(jpe?g|gif|bit|tiff|ppm)'
arg isfile $0
plumb to image
plumb client page -wi
```

(Lines beginning with `#` are commentary.) Consider how this rule handles the following message, annotated down the left column for clarity:

```

src      plumbtest
dst
wdir     /usr/rob/pics
type     text
attr
ndata    9
data     horse.gif

```

The `is` verb specifies a precise match, and the `type` field of the message is the string `text`, so the first pattern succeeds. The `matches` verb invokes a regular expression pattern match of the object (here `data`) against the argument pattern. Both `matches` patterns in this rule set will succeed, and in the process set the variables `$0` to the matched string, `$1` to the first parenthesized submatch, and so on (analogous to `&`, `\1`, etc. in `ed`'s regular expressions). The pattern `arg isfile $0` verifies that the named file, `horse.gif`, is an actual file in the directory `/usr/rob/pics`. If all the patterns succeed, one of the actions will be executed.

There are two actions in this rule set. The `plumb to` rule specifies `image` as the destination port of the message. By convention, the plumber mounts its services in the directory `/mnt/plumb`, so in this case if the file `/mnt/plumb/image` has been opened, the message will be made available to the program reading from it. Note that the message does not name a port, but the rule set that matches the message does, and that is sufficient to dispatch the message. If on the other hand a message matches no rule but has an explicit port mentioned, that too is sufficient.

If no client has opened the `image` port, that is, if the program `page` is not already running, the `plumb client` action gives the execution script to start the application and send the message on its way; the `-wi` arguments tell `page` to create a window and to receive its initial arguments from the plumbing port. The process by which the plumber starts a program is described in more detail in the next section.

It may seem odd that there are two `matches` rules in this example. The reason is related to the way the plumber can use the rules themselves to refine the `data` in the message, somewhat in the manner of Structural Regular Expressions [Pike87a]. For example, consider what happens if the cursor is at the last character of

```
% make nightmare>horse.gif
```

and the user asks to `plumb` what the cursor is pointing at. The program creating the plumbing message—in this case the terminal emulator running the window—can send the entire white-space-delimited string `nightmare>horse.gif` or even the entire line, and the combination of `matches` rules can determine that

the user was referring to the string `horse.gif`. The user could of course select the entire string `horse.gif`, but it's more convenient just to point in the general location and let the machine figure out what should be done. The process is as follows.

The application generating the message adds a special attribute to the message, named `click`, whose numerical value is the offset of the cursor—the selection point—within the data string. This attribute tells the plumber two things: first, that the regular expressions in `matches` rules should be used to identify the relevant data; and second, approximately where the relevant data lies. The plumber will then use the first `matches` pattern to identify the longest leftmost match that touches the cursor, which will extract the string `horse.gif`, and the second pattern will then verify that that names a picture file. The rule set succeeds and the data is winnowed to the matching substring before being sent to its destination.

Each `matches` pattern within a given rule set must match the same portion of the string, which guarantees that the rule set fails to match a string for which the second pattern matches only a portion. For instance, our example rule set should not execute if the data is the string `horse.gif.`, and although the first pattern will match `horse.gif.`, the second will match only `horse.gif` and the rule set will fail.

The same approach of multiple `matches` rules can be used to exclude, for instance, a terminal period from a file name or URL, so a file name or URL at the end of a sentence is recognized properly.

If a `click` attribute is not specified, all patterns must match the entire string, so the user has an option: he or she may select exactly what data to send, or may instead indicate where the data is by clicking the selection button on the mouse and letting the machine locate the URL or image file name within the text. In other words, the user can control the contents of the message precisely when required, but the default, simplest action in the user interface does the right thing most of the time.

## 5. How Messages are Handled in the Plumber

An application creates a message header, fills in whatever fields it wishes to define, attaches the data, and writes the result to the file `send` in the plumber's service directory, `/mnt/plumb`. The plumber receives the message and applies the plumbing rules successively to it. When a rule set matches, the message is dispatched as indicated by that rule set and processing continues with the next message. If no rule set matches the message, the plumber indicates this by returning a

write error to the application, that is, the write to `/mnt/plumb/send` fails, with the resulting error string describing the failure. (Plan 9 uses strings rather than pre-defined numbers to describe error conditions.) Thus a program can discover whether a plumbing message has been sent successfully.

After a matching rule set has been identified, the plumber applies a series of rewriting steps to the message. Some rewritings are defined by the rule set; others are implicit. For example, if the message does not specify a destination port, the outgoing message will be rewritten to identify it. If the message does specify the port, the rule set will only match if any `plumb` to action in the rule set names the same port. (If it matches no rule sets, but mentions a port, it will be sent there unmodified.)

The rule set may contain actions that explicitly rewrite components of the message. These may modify the attribute list or replace the data section of the message. Here is a sample rule set that does both. It matches strings of the form `plumb.h` or `plumb.h:27`. If that string identifies a file in the standard C include directory, `/sys/include`, perhaps with an optional line number, the outgoing message is rewritten to contain the full path name and an attribute, `addr`, to hold the line number:

```
# .h files found in /sys/include
# are passed to edit
type is text
data matches '([a-zA-Z0-9]+\.[h])
(:([0-9]+))?'
arg isfile /sys/include/$1
data set /sys/include/$1
attr add addr=$3
plumb to edit
```

The data set rule replaces the contents of the data, and the `attr add` rule adds a new attribute to the message. The intent of this rule is to permit one to plumb an include file name in a C program to trigger the opening of that file, perhaps at a specified line, in the text editor. A variant of this rule, discussed below, tells the editor how to interpret syntax errors from the compiler, or the output of `grep -n`, both of which use a fixed syntax `file:line` to identify a line of source.

The Plan 9 text editors interpret the `addr` attribute as the definition of which portion of the file to display. In fact, the real rule includes a richer definition of the address syntax, so one may plumb strings such as `plumb.h:/plumbsend` (using a regular expression after the `/`) to pop up the declaration of a function in a C header file.

Another form of rewriting is that the plumber may modify the attribute list of the message to clarify how to

handle the message. The primary example of this involves the treatment of the `click` attribute, described in the previous section. If the message contains a `click` attribute and the matching rule set uses it to extract the matching substring from the data, the plumber deletes the `click` attribute and replaces the data with the matching substring.

Once the message is rewritten, the actions of the matching rule set are examined. If the rule set contains a `plumb to` action and the corresponding port is open—that is, if a program is already reading from that port—the message is delivered to the port. The application will receive the message and handle it as it sees fit. If the port is not open, a `plumb start` or `plumb client` action will start a new program to handle the message.

The `plumb start` action is the simpler: its argument specifies a command to run instead of passing on the message; the message is discarded. Here for instance is a rule that, given the process id (pid) of an existing process, starts the `acid` debugger [Wint94] in a new window to examine that process:

```
# processes go to acid
# (assuming strlen(pid) >= 2)
type is text
data matches '[a-zA-Z0-9.:\-/_]+'
data matches '[0-9][0-9]+'
arg isdir /proc/$0
plumb start window acid $0
```

(Note the use of multiple matches rules to avoid misfires from strings like `party.1999`.) The `arg isdir` rule checks that the pid represents a running process (or broken one; Plan 9 does not create core files but leaves broken processes around for debugging) by checking that the process file system has a directory for that pid [Kill84]. Using this rule, one may plumb the pid string printed by the `ps` command or by the operating system when the program breaks; the debugger will then start automatically.

The other startup action, `plumb client`, is used when a program will read messages from the plumbing port. For example, text editors can read files specified as command arguments, so one could use a `plumb start` rule to begin editing a file. If, however, the editor will read messages from the `edit` plumbing port, letting it read the message from the port insures that it uses other information in the message, such as the line number to display. The `plumb client` action is therefore like `plumb start`, but keeps the message around for delivery when the application opens the port. Here is the full rule set to pass a regular file to the text editor:

```
# existing files, possibly tagged by address, go to editor
type is text
data matches '([a-zA-Z0-9_/\-]*[a-zA-Z0-9_/\-])('$addr')?'
arg isfile $1
data set $1
attr add addr=$3
plumb to edit
plumb client window $editor
```

If the editor is already running, the `plumb to rule` causes it to receive the message on the port. If not, the command `'window $editor'` will create a new window (using the Plan 9 program `window`) to run the editor, and once that starts it will open the `edit` plumbing port as usual and discover this first message already waiting.

The variables `$editor` and `$addr` in this rule set are macros defined in the plumbing rules file; they specify the name of the user's favorite text editor and a regular expression that matches that editor's address syntax, such as line numbers and patterns. This rule set lives in a library of shared plumbing rules that users' private rules can build on, so the rule set needs to be adaptable to different editors and their address syntax. The macro definitions for Acme and Sam [Pike94,Pike87b] look like this:

```
editor=acme
# or editor=sam
addrelem='((#[0-9]+) |
    (/ [A-Za-z0-9_^\^]+/?) | [.$])'
addr=:( $addrelem( [, ; + \- ] $addrelem ) *)
```

Finally, the application reads the message from the appropriate port, such as `/mnt/plumb/edit`, unpacks it, and goes to work.

## 6. Message Delivery

In summary, a message is delivered by writing it to the `send` file and having the plumber, perhaps after some rewriting, send it to the destination port or start a new application to handle it. If no destination can be found by the plumber, the original write to the `send` file will fail, and the application will know the message could not be delivered.

If multiple applications are reading from the destination port, each will receive an identical copy of the message; that is, the plumber implements fan-out. The number of messages delivered is equal to the number of clients that have opened the destination port. The plumber queues the messages and makes sure that each application that opened the port before the message was written gets exactly one copy.

This design minimizes blocking in the sending applications, since the write to the `send` file can complete as

soon as the message has been queued for the appropriate port. If the plumber waited for the message to be read by the recipient, the sender could block unnecessarily. Unfortunately, this design also means that there is no way for a sender to know when the message has been handled; in fact, there are cases when the message will not be delivered at all, such as if the recipient exits while there are still messages in the queue. Since the plumber is part of a user interface, and not an autonomous message delivery system, the decision was made to give the non-blocking property priority over reliability of message delivery. In practice, this tradeoff has worked out well: applications almost always know when a message has failed to be delivered (the `write` fails because no destination could be found), and those occasions when the sender believes incorrectly that the message has been delivered are both extremely rare and easily recognized by the user—usually because the recipient application has exited.

## 7. The Rules File

The plumber begins execution by reading the user's startup plumbing rules file, `lib/plumbing`. Since the plumber is implemented as a file server, it can also present its current rules as a dynamic file, a design that provides an easily understood way to maintain the rules.

The file `/mnt/plumb/rules` is the text of the rule set the plumber is currently using, and it may be edited like a regular file to update those rules. To clear the rules, truncate that file; to add a new rule set, append to it:

```
% echo 'type is text
data is self-destruct
plumb start rm -rf $HOME' >>
    /mnt/plumb/rules
```

This rule set will take effect immediately. If it has a syntax error, the write will fail with an error message from the plumber, such as `'malformed rule'` or `'undefined verb'`.

To restore the plumber to its startup configuration,

```
% cp /usr/$user/lib/plumbing \
    /mnt/plumb/rules
```

For more sophisticated changes, one can of course use a regular text editor to modify `/mnt/plumb/rules`.



This simple way of maintaining an active service could profitably be adopted by other systems. It avoids the need to reboot, to update registries with special tools, or to send asynchronous signals to critical programs.

## 8. The User Interface

One unusual property of the plumbing system is that the user interface that programs provide to access it can vary considerably, yet the result is nonetheless a unifying force in the environment. Shells talk to editors, image viewers, and web browsers; debuggers talk to editors; editors talk to themselves; and the window system talks to everybody.

The plumber grew out of some of the ideas of the Acme editor/window-system/user interface [Pike94], in particular its ‘acquisition’ feature. With a three-button mouse, clicking the right button in Acme on a piece of text tells Acme to get the thing being pointed to. If it is a file name, open the file; if it is a directory, open a viewer for its contents; if a line number, go to that line; if a regular expression, search for it. This one-click access to anything describable textually was very powerful but had several limitations, of which the most important were that Acme’s rules for interpreting the text (that is, the implicit hyperlinks) were hard-wired and inflexible, and that they only applied to and within Acme itself. One could not, for example, use Acme’s power to open an image file, since Acme is a text-only system.

The plumber addresses these limitations, even with Acme itself: Acme now uses the plumber to interpret the right button clicks for it. When the right button is clicked on some text, Acme constructs a plumbing message much as described above, using the `click` attribute and the white-space-delimited text surrounding the click. It then writes the message to the plumber; if the write succeeds, all is well. If not, it falls back to its original, internal rules, which will result in a context search for the word within the current document.

If the message is sent successfully, the recipient is likely to be Acme itself, of course: the request may be to open a file, for example. Thus Acme has turned the plumber into an external component of its own operation, while expanding the possibilities; the operation might be to start an image viewer to open a picture file, something Acme cannot do itself. The plumber expands the power of Acme’s original user interface.

Traditional menu-driven programs such as the text editor Sam [Pike87b] and the default shell window of the

window system 8½ [Pike91] cannot dedicate a mouse button solely to plumbing, but they can certainly dedicate a menu entry. The editing menu for such programs now contains an entry, `plumb`, that creates a plumbing message using the current selection. (Acme manages to send a message by clicking on the text with one button; other programs require a click with the select button and then a menu operation.) For example, after this happens in a shell window:

```
% make
cc -c shaney.c
shaney.c:232: i undefined
...
```

one can click anywhere on the string `shaney.c:232`, execute the `plumb` menu entry, and have line 232 appear in the text editor, be it Sam or Acme—whichever has the `edit` port open. (If this were an Acme shell window, it would be sufficient to right-click on the string.)

[An interesting side line is how the window system knows what directory the shell is running in; in other words, what value to place in the `wdir` field of the `plumb` message. Recall that 8½ is, like many Plan 9 programs, a file server. It now serves a new file, `/dev/wdir`, that is private to each window. Programs, in particular the Plan 9 shell, `rc`, can write that file to inform the window system of its current directory. When a `cd` command is executed in an interactive shell, `rc` updates the contents of `/dev/wdir` and plumbing can proceed with local file names.]

Of course, users can plumb image file names, process ids, URLs, and other items—any string whose syntax and disposition are defined in the plumbing rules file. An example of how the pieces fit together is the way Plan 9 now handles mail, particularly MIME-encoded messages.

When a new mail message arrives, the mail receiver process sends a plumbing message to the `newmail` port, which notifies any interested process that new mail is here. The plumbing message contains information about the mail, including its sender, date, and current location in the file system. The interested processes include a program, `faces`, that gives a graphical display of the mail box using faces to represent the senders of messages [PiPr85], as well as interactive mail programs such as the Acme mail viewer [Pike94]. The user can then click on the face that appears, and the `faces` program will send another plumbing message, this time to the `showmail` port. Here is the rule for that port:

```
% ls -l /mail/fs/mbox/25
d-r-xr-xr-x M 20 rob rob      0 Nov 21 13:06 /mail/fs/mbox/25/1
d-r-xr-xr-x M 20 rob rob      0 Nov 21 13:06 /mail/fs/mbox/25/2
--r--r--r-- M 20 rob rob 28678 Nov 21 13:06 /mail/fs/mbox/25/body
--r--r--r-- M 20 rob rob      0 Nov 21 13:06 /mail/fs/mbox/25/cc
...
% mail
25 messages
: 25
From: presotto
Date: Sun Nov 21 13:05:51 EST 1999
To: rob

Check this out.

==> 2/ (image/jpeg) [inline]
      /mail/fs/mbox/25/2/fabio.jpg
:
```

Figure 2. A terminal session illustrating the mail file system.

```
# faces -> new mail window for message
type is text
data matches '[a-zA-Z0-9_\-./]+'
data matches '/mail/fs/[a-zA-Z0-9/]+/[0-9]+'
plumb to showmail
plumb start window edmail -s $0
```

If a program, such as the Acme mail reader, is reading that port, it will open a new window in which to display the message. If not, the `plumb start` rule will create a new window and run `edmail`, a conventional mail reading process, to examine it. Notice how the plumbing connects the components of the interface together the same way regardless of which components are actually being used to view mail.

There is more to the mail story. Naturally, mail boxes in Plan 9 are treated as little file systems, which are synthesized on demand by a special-purpose file server that takes a flat mail box file and converts it into a set of directories, one per message, with component files containing the header, body, MIME information, and so on. Multi-part MIME messages are unpacked into multi-level directories as shown in Figure 2.

Since the components are all (synthetic) files, the user can plumb the pieces to view embedded pictures, URLs, and so on. Note that the mail program can plumb the contents of inline attachments automatically, without user interaction; in other words, plumbing lets the mailer handle multimedia data without itself interpreting it.

At a more mundane level, a shell command, `plumb`, can be used to send messages:

```
% cd /usr/rob/src
% plumb mem.c
```

will send the appropriate message to the `edit` port. A surprising use of the `plumb` command is in actions

within the plumbing rules file. In our lab, we commonly receive Microsoft Word documents by mail, but we do not run Microsoft operating systems on our machines so we cannot view them without at least rebooting. Therefore, when a Word document arrives in mail, we could plumb the `.doc` file but the text editor could not decode it. However, we have a program, `doc2txt`, that decodes the Word file format to extract and format the embedded text. The solution is to use `plumb` in a `plumb start` action to invoke `doc2txt` on `.doc` files and synthesize a plain text file:

```
# rule set for MS Word documents
type is text
data matches '[a-zA-Z0-9_\-./]+'
data matches '([a-zA-Z0-9_\-./]+)\.doc'
arg isfile $0
plumb start doc2txt $data | \
    plumb -i -d edit \
    -a action=showdata \
    -a filename=$0
```

The arguments to `plumb` tell it to take standard input as its data rather than the text of the arguments (`-i`), define the destination port (`-d edit`), and set a conventional attribute so the editor knows to show the message data itself rather than interpret it as a file name (`-a action=showdata`) and provide the original file name (`-a filename=$0`). Now when a user plumbs a `.doc` file the plumbing rules run a process to extract the text and send it as a temporary file to the editor for viewing. It's imperfect, but it's easy and it beats rebooting.

Another simple example is a rule that turns man pages into hypertext. Manual page entries of the form `plumber(1)` can be clicked on to pop up a window containing the formatted 'man page'. That man page will in turn contain more such citations, which will also

be clickable. The rule is a little like that for Word documents:

```
# man index entries are synthesized
type is text
data matches '([a-zA-Z0-9_\-.\/]+)
\([([0-9])\))\)'
plumb start man $2 $1 | \
    plumb -i -d edit \
    -a action=showdata \
    -a filename=/man/$1($2)
```

There are many other inventive uses of plumbing. One more should give some of the flavor. We have a shell script, `src`, that takes as argument the name of an executable binary file. It examines the symbol table of the binary to find the source file from which it was compiled. Since the Plan 9 compilers place full source path names in the symbol table, `src` can discover the complete file name. That is then passed to `plumb`, complete with the line number to find the symbol `main`. For example,

```
% src plumb
```

is all it takes to pop up an editor window on the main routine of the `plumb` command, beginning at line 39 of `/sys/src/cmd/plumb/plumb.c`. Like most uses of plumbing, this is not a breakthrough in functionality, but it is a great convenience.

## 9. Why This Architecture?

The design of the plumbing system is peculiar: a centralized language-based file server does most of the work, while compared to other systems the applications themselves contribute relatively little. This architecture is deliberate, of course.

That the plumber's behavior is derived from a linguistic description gives the system great flexibility and dynamism—rules can be added and changed at will, without rebooting—but the existence of a central library of rules ensures that, for most users, the environment behaves in well-established ways.

That the plumber is a file server is perhaps the most unusual aspect of its design, but is also one of the most important. Messages are passed by regular I/O operations on files, so no extra technology such as remote procedure call or request brokers needs to be provided; messages are transmitted by familiar means. Almost every service in Plan 9 is a file server, so services can be exported trivially using the system's remote file system operations [Pike93]. The plumber is no exception; plumbing messages pass routinely across the network to remote applications without any special provision, in contrast to some commercial IPC mechanisms that become significantly more complex when they involve

multiple machines. As I write this, my window system is talking to applications running on three different machines, but they all share a single instance of the plumber and so can interoperate to integrate my environment. Plan 9 uses a shared file name space to combine multiple networked machines—compute servers, file servers, and interactive workstations—into a single computing environment; plumbing's design as a file server is a natural by-product of, and contributor to, the overall system architecture [Pike92].

The centrality of the plumber is also unusual. Other systems tend to let the applications determine where messages will go; consider mail readers that recognize and highlight URLs in the messages. Why should just the mail readers do this, and why should they just do it for URLs? (Acme was guilty of similar crimes.) The plumber, by removing such decisions to a central authority, guarantees that all applications behave the same and simultaneously frees them all from figuring out what's important. The ability for the plumber to excerpt useful data from within a message is critical to the success of this model.

The entire system is remarkably small. The plumber itself is only about two thousand lines of C code. Most applications work fine in a plumbing environment without knowing about it at all; some need trivial changes such as to standardize their error output; a few need to generate and receive plumbing messages. But even to add the ability to send and receive messages in a program such as text editor is short work, involving typically a few dozen lines of code. Plumbing fits well into the existing environment.

But plumbing is new and it hasn't been pushed far enough yet. Most of the work so far has been with textual messages, although the underlying system is capable of handling general data. We plan to reimplement some of the existing data movement operations, such as cut and paste or drag and drop, to use plumbing as their exchange mechanism. Since the plumber is a central message handler, it is an obvious place to store the 'clipboard'. The clipboard could be built as a special port that holds onto messages rather than deleting them after delivery. Since the clipboard would then be holding a plumbing message rather than plain text, as in the current Plan 9 environment, it would become possible to cut and paste arbitrary data without providing new mechanism. In effect, we would be providing a new user interface to the existing plumbing facilities.

Another possible extension is the ability to override plumbing operations interactively. Originally, the plan was to provide a mechanism, perhaps a pop-up menu, that one could use to direct messages, for example to



send a PostScript file to the editor rather than the PostScript viewer by naming an explicit destination in the message. Although this deficiency should one day be addressed, it should be done without complicating the interface for invoking the default behavior. Meanwhile, in practice the default behavior seems to work very well in practice—as it must if plumbing is to be successful—so the lack of overrides is not keenly felt.

## 10. Comparison with Other Systems

The ideas of the plumbing system grew from an attempt to generalize the way Acme acquires files and data. Systems further from that lineage also share some properties with plumbing. Most, however, require explicit linking or message passing rather than plumbing's implicit, context-based pattern matching, and none has the plumber's design of a language-based file server.

Reiss's FIELD system [Reis95] probably comes the closest to providing the facilities of the plumber. It has a central message-passing mechanism that connects applications together through a combination of a library and a pattern-matching central message dispatcher that handles message send and reply. The main differences between FIELD's message dispatcher and the plumber are first that the plumber is based on a special-purpose language while the FIELD system uses an object-oriented library, second that the plumber has no concept of a reply to a message, and finally that the FIELD system has no concept of port. But the key distinction is probably in the level of use. In FIELD, the message dispatcher is a critical integrating force of the underlying programming environment, handling everything from debugging events to changing the working directory of a program. Plumbing, by contrast, is intended primarily for integrating the user interface of existing tools; it is more modest and very much simpler. The central advantage of the plumber is its convenience and dynamism; the FIELD system does not share the ease with which message dispatch rules can be added or modified.

The inspiration for Acme was the user interface to the object-oriented Oberon system [WiGu92]. Oberon's user interface interprets mouse clicks on strings such as `Obj.meth` to invoke calls to the method `meth` of the object `Obj`. This was the starting point for Acme's middle-button execution [Pike94], but nothing in Oberon is much like Acme's right-button 'acquisition', which was the starting point for the plumber. Oberon's implicit method-based linking is not nearly as general as the pattern-matched linking of the plumber, nor does its style of user-triggered method call correspond well to the more general idea of inter-application communication of plumbing messages.

Microsoft's OLE interface is another relative. It allows one application to *embed* its own data within another's, for example to place an Excel spreadsheet within a Frame document; when Frame needs to format the page, it will start Excel itself, or at least some of its DLLs, to format the spreadsheet. OLE data can only be understood by the application that created it; plumbing messages, by contrast, contain arbitrary data with a rigidly formatted header that will be interpreted by the pattern matcher and the destination application. The plumber's simplified message format may limit its flexibility but makes messages easy and efficient to dispatch and to interpret. At least for the cut-and-paste style of exchange OLE encourages, plumbing gives up some power in return for simplicity, while avoiding the need to invoke a vestigial program (if Excel can be called a vestige) every time the pasted data is examined. Plumbing is also better suited to other styles of data exchange, such as connecting compiler errors to the text editor.

The Hyperbole [Wein] package for Emacs adds hypertext facilities to existing documents. It includes explicit links and, like plumbing, a rule-driven way to form implicit links. Since Emacs is purely textual, like Acme, Hyperbole does not easily extend to driving graphical applications, nor does it provide a general interprocess communication method. For instance, although Hyperbole provides some integration for mail applications, it cannot provide the glue that allows a click on a face icon in an external program to open a mail message within the viewer. Moreover, since it is not implemented as a file server, Hyperbole does not share the advantages of that architecture.

Henry's `error` program in 4BSD echoes a small but common use of plumbing. It takes the error messages produced by a compiler and drives a text editor through the steps of looking at each one in turn; the notion is to quicken the compile/edit/debug cycle. Similar results are achieved in EMACS by writing special M-LISP macros to parse the error messages from various compilers. Although for this particular purpose they may be more convenient than plumbing, these are specific solutions to a specific problem and lack plumbing's generality.

Of course, the resource forks in MacOS and the association rules for file name extensions in Windows also provide some of the functionality of the plumber, although again without the generality or dynamic nature.

Closer to home, Ousterhout's Tcl (Tool Command Language) [Oust90] was originally designed to embed a little command interpreter in each application to control interprocess communication and provide a level of inte-



gration. Plumbing, on the other hand, provides minimal support within the application, offloading most of the message handling and all the command execution to the central plumber.

The most obvious relative to plumbing is perhaps the hypertext links of a web browser. Plumbing differs by synthesizing the links on demand. Rather than constructing links within a document as in HTML, plumbing uses the context of a button click to derive what it should link to. That the rules for this decision can be modified dynamically gives it a more fluid feel than a standard web browsing world. One possibility for future work is to adapt a web browser to use plumbing as its link-following engine, much as Acme used plumbing to offload its acquisition rules. This would connect the web browser to the existing tools, rather than the current trend in most systems of replacing the tools by a browser.

Each of these prior systems—and there are others, e.g. [Pasa93, Free93]—addresses a particular need or subset of the issues of system integration. Plumbing differs because its particular choices were different. It focuses on two key issues: centralizing and automating the handling of interprocess communication among interactive programs, and maximizing the convenience (or minimizing the trouble) for the human user of its services. Moreover, the plumber's implementation as a file server, with messages passed over files it controls, permits the architecture to work transparently across a network. None of the other systems discussed here integrates distributed systems as smoothly as local ones without the addition of significant extra technology.

## 11. Discussion

There were a few surprises during the development of plumbing. The first version of plumbing was done for the Inferno system [Dorw97a,Dorw97b], using its file-to-channel mechanism to mediate the IPC. Although it was very simple to build, it encountered difficulties because the plumber was too disconnected from its clients; in particular, there was no way to discover whether a port was in use. When plumbing was implemented afresh for Plan 9, it was provided through a true file server. Although this was much more work, it paid off handsomely. The plumber now knows whether a port is open, which makes it easy to decide whether a new program must be started to handle a message, and the ability to edit the rules file dynamically is a major advantage. Other advantages arise from the file-server design, such as the ease of exporting plumbing ports across the network to remote machines and the implicit security model a file-based interface provides: no one has permission to open my private plumbing files.

On the other hand, Inferno was an all-new environment and the user interface for plumbing was able to be made uniform for all applications. This was impractical for Plan 9, so more *ad hoc* interfaces had to be provided for that environment. Yet even in Plan 9 the advantages of efficient, convenient, dynamic interprocess communication outweigh the variability of the user interface. In fact, it is perhaps a telling point that the system works well for a variety of interfaces; the provision of a central, convenient message-passing service is a good idea regardless of how the programs use it.

Plumbing's rule language uses only regular expressions and a few special rules such as `isfile` for matching text. There is much more that could be done. For example, in the current system a JPEG file can be recognized by a `.jpg` suffix but not by its contents, since the plumbing language has no facility for examining the *contents* of files named in its messages. To address this issue without adding more special rules requires rethinking the language itself. Although the current system seems a good balance of complexity and functionality, perhaps a richer, more general-purpose language would permit more exotic applications of the plumbing model.

In conclusion, plumbing adds an effective, easy-to-use inter-application communication mechanism to the Plan 9 user interface. Its unusual design as a language-driven file server makes it easy to add context-dependent, dynamically interpreted, general-purpose hyperlinks to the desktop, for both existing tools and new ones.

## 12. Acknowledgements

Dave Presotto wrote the mail file system and `edmail`. He, Russ Cox, Sape Mullender, and Cliff Young influenced the design, offered useful suggestions, and suffered early versions of the software. They also made helpful comments on this paper, as did Dennis Ritchie and Brian Kernighan.

## 13. References

- [Dorw97a] Sean Dorward, Rob Pike, David Leo Presotto, Dennis M. Ritchie, Howard W. Trickey, and Philip Winterbottom, "Inferno", *Proceedings of the IEEE Comcon 97 Conference*, San Jose, 1997, pp. 241-244.
- [Dorw97b] Sean Dorward, Rob Pike, David Leo Presotto, Dennis M. Ritchie, Howard W. Trickey, and Philip Winterbottom, "The Inferno Operating System", *Bell Labs Technical Journal*, 2, 1, Winter, 1997.
- [Free93] FreeBSD, Syslog configuration file manual `syslog.conf(0)`.

- [Kill84] T. J. Killian, "Processes as Files", *Proceedings of the Summer 1984 USENIX Conference*, Salt Lake City, 1984, pp. 203-207.
- [Oust90] John K. Ousterhout, "Tcl: An Embeddable Command Languages", *Proceedings of the Winter 1990 USENIX Conference*, Washington, 1990, pp. 133-146.
- [Pasa93] Vern Paxson and Chris Saltmarsh, "Glish: A User-Level Software Bus for Loosely-Coupled Distributed Systems", *Proceedings of the Winter 1993 USENIX Conference*, San Diego, 1993, pp. 141-155.
- [Pike87a] Rob Pike, "Structural Regular Expressions", *EUUG Spring 1987 Conference Proceedings*, Helsinki, May 1987, pp. 21-28.
- [Pike87b] Rob Pike, "The Text Editor sam", *Software - Practice and Experience*, 17, 5, Nov. 1987, pp. 813-845.
- [Pike91] Rob Pike, "8½, the Plan 9 Window System", *Proceedings of the Summer 1991 USENIX Conference*, Nashville, 1991, pp. 257-265.
- [Pike93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom, "The Use of Name Spaces in Plan 9", *Operating Systems Review*, 27, 2, April 1993, pp. 72-76.
- [Pike94] Rob Pike, "Acme: A User Interface for Programmers", *Proceedings of the Winter 1994 USENIX Conference*, San Francisco, 1994, pp. 223-234.
- [PiPr85] Rob Pike and Dave Presotto, "Face the Nation", *Proceedings of the USENIX Summer 1985 Conference*, Portland, 1985, pg. 81.
- [Reis95] Steven P. Reiss, *The FIELD Programming Environment: A Friendly Integrated Environment for Learning and Development*, Kluwer, Boston, 1995.
- [Wein] Bob Weiner, *Hyperbole User Manual*, [http://www.cs.indiana.edu/elisp/hyperbole/hyperbole\\_1.html](http://www.cs.indiana.edu/elisp/hyperbole/hyperbole_1.html).
- [Wint94] Philip Winterbottom, "ACID: A Debugger based on a Language", *Proceedings of the USENIX Winter Conference*, San Francisco, CA, 1994.
- [WiGu92] Niklaus Wirth and Jurg Gutknecht, *Project Oberon: The Design of an Operating System and Compilers*, Addison-Wesley, Reading, 1992.

# Integrating a Command Shell Into a Web Browser

Robert C. Miller and Brad A. Myers

*Carnegie Mellon University*

{rcm,bam}@cs.cmu.edu

## Abstract

The transition from command-line interfaces to graphical interfaces has resulted in programs that are easier to learn and use, but harder to automate and reuse. Another transition is now underway, to HTML interfaces hosted by a web browser. To help users automate HTML interfaces, we propose the *browser-shell*, a web browser that integrates a command interpreter into the browser's Location box. The browser-shell's command language is designed for extracting and manipulating HTML and text, and commands can also invoke local programs. Command input is drawn from the current browser page, and command output is displayed as a new page. The browser-shell brings to web browsing many advantages of the Unix shell, including scripting web services and creating pipelines of web services and local programs. A browser-shell also allows legacy command-line programs to be wrapped with an HTML/CGI interface that is graphical but still scriptable, and offers a new shell interaction model, different from the conventional type-script model, which may improve usability in some respects.

## 1 Introduction

The transition from command-line interfaces to graphical interfaces carries with it a significant cost. In the Unix shell, for example, programs accept plain text as input and generate plain text as output. This makes it easy to write scripts that automate user interaction. An expert Unix user can create sophisticated programs on the spur of the moment, by hooking together simpler programs with pipelines and command substitution. For example:

```
kill `ps ax | grep xterm | awk '{print $1;}'`
```

This command uses `ps` to list information about running processes, `grep` to find just the `xterm` processes, `awk` to select just the process identifiers, and finally `kill` to kill those processes.

These capabilities are lost in the transition to a graphical user interface (GUI). GUI programs accept mouse clicks

and keystrokes as input and generate raster graphics as output. Automating graphical interfaces is hard, unfortunately, because mouse clicks and pixels are too low-level for effective automation and interprocess communication. Attempts to introduce Unix shell features like pipelining into graphical user interfaces [3, 6, 7, 8, 15, 16] have been unsuccessful, largely because they were not integrated well with existing applications, required extra work from application developers to expose hooks and programming interfaces, or were too hard to use.

With the advent of the World Wide Web, another transition is underway, this time to distributed web applications that run on a web server and interact with the user through a web browser. Most web services accept input from HTML forms and generate output as HTML pages. Since HTML is textual and capable of being parsed and manipulated, we have the opportunity to recover some of the interactive automation capabilities that were available in the Unix shell, but missing in graphical interfaces. Consider the following web-browsing tasks that could be partially or totally automated:

- Download and print a group of links on a page;
- Compare airfares and schedules for several choices of departure and arrival dates;
- Look up a colleague in the online university phone-book, obtain a home address, locate the address on a map, and get driving directions;
- Given a list of books to read, search for each book in the local library catalog, and if the book is not on the shelves, buy it from an online bookstore;
- Make a smart alarm clock that announces the current temperature from an online weather report, and the time until the next bus departs from an online schedule, while you dress in the morning.

As a step towards automating these tasks and others, we have extended a web browser in several ways:

1. *Embedding a pattern language for matching text and HTML, and a suite of text-processing tools for*

*extracting and manipulating web page data.* High-level pattern-matching and text manipulation are essential to web automation, acting as a *glue language* for connecting unrelated web services and programs.

2. *Embedding a scripting language and integrating a command interpreter into the Location box.* In addition to accepting a typed URL, the browser window's Location box can also accept a typed command with arguments. A command may be a built-in command, a user-defined script, or an external program. The built-in scripting language includes commands for automatic web browsing, such as clicking on hyperlinks, filling out forms, and extracting data from web pages.
3. *Using the browser window to display command output and construct pipelines of commands.* When a command is invoked, it takes its input from the current page in the browser window, and sends its output back to the browser window as a new page.
4. *Including executed commands in the browsing history.* Forward and Back navigate through command output pages as well as web pages. Part of the history can be extracted and saved as a script for later execution.

We have implemented these extensions in a prototype web browser named LAPIS (Lightweight Architecture for Processing Information Structure). The first extension, consisting of a pattern language and text-processing tools, was described in a previous paper [14], which is summarized below. This paper focuses on the other three features, which integrate a command shell into the web browser to create a *browser-shell*.

The browser-shell addresses the problem of interactive web automation by allowing the user to apply patterns, script commands, and external programs directly to the browser page. For one-shot tasks, commands can be interleaved with manual browsing to perform the task as quickly and directly as possible. For repeated tasks, the user can interactively define a script by invoking a sequence of commands on example data, using the Back button to correct mistakes, and then copying the command sequence out of the browsing history and saving it as a script.

The browser-shell concept has implications beyond web automation, two of which are considered in this paper:

1. *HTML interfaces for local programs.* Currently, programs with HTML interfaces must be installed

in a web server in order to handle form submissions. LAPIS can submit forms to local programs by the Common Gateway Interface (CGI) [17], an existing standard used by web servers. This opens the possibility of running HTML applications entirely locally. HTML offers benefits of both a graphical user interface (GUI) and a command-line interface (CLI). An HTML interface can be as easy to learn and use as a GUI, yet still open to automation like a CLI. As a demonstration, we have wrapped an HTML interface around the Unix *find* program.

2. *Using the browser as a command shell, in place of the Unix shell or MS-DOS command prompt.* The browser-shell can be used to invoke local programs, but it behaves differently from a conventional typescript shell. Whereas a typescript shell interleaves commands with program output in the same window, a browser-shell displays commands and program output in separate parts of the browser window, and automatically redirects a program's input from the current page. These differences make some tasks easier, such as viewing program output and constructing pipelines, but others harder, such as running legacy programs that use standard input to interact with the user. The tradeoffs are discussed in more detail in section 5.

The remainder of this paper is organized as follows. Section 2 covers related work. Section 3 describes important features of the LAPIS browser-shell, including the pattern language, the scripting language, and invocation of external programs. Section 4 describes our prototype implementation of LAPIS and contrasts some implementation alternatives. Section 5 discusses some of the implications of integrating a command shell into a web browser, in particular creating local programs with HTML interfaces and using the browser as an alternative interface to the system command prompt. Section 6 reports on the status of the LAPIS prototype, and Section 7 concludes.

## 2 Related Work

Several systems have addressed the problem of web automation. One approach is *macro recording*, typified by LiveAgent [11]. LiveAgent automates a task by recording a sequence of browsing actions in Netscape through a local HTTP proxy. Macro recording requires little learning on the part of the user, but recorded macros suffer from limited expressiveness, often lacking variables, conditionals, and iteration.



Another approach is *scripting*, writing a program in a scripting language such as Perl, Tcl, or Python. These scripting languages are fully expressive, Turing-complete programming languages, but programs written in these languages must be developed, tested, and invoked outside the web browser, making them difficult to incorporate into a web user's work flow. The overhead of switching to an external scripting language tends to discourage the kind of spur-of-the-moment automation required by the tasks described above, in which interactive operations might be mixed with automation in order to finish a task more quickly.

A particularly relevant scripting language is WebL [9], which provides high-level *service combinators* for invoking web services and a *markup algebra* similar to the LAPIS pattern language for extracting results. Like other scripting languages, WebL lacks tight integration with a web browser, forcing a user to study the HTML source of a web service to develop markup patterns and reverse-engineer form interfaces. In LAPIS, web automation can be done while viewing rendered web pages in the browser, and simple tasks can be automated entirely by demonstrating the steps on examples.

Other systems have tackled more restricted forms of web automation by demonstration. Turquoise [13] and Internet Scrapbook [22] construct a *personalized newspaper*, a dynamic collage of pieces clipped from other web pages, by generalizing from a cut-and-paste demonstration. SPHINX [12] creates a web crawler by demonstration, learning which URLs to follow from positive and negative examples.

Wrapping GUI frontends around CLI programs is a common way to support both ease-of-use and scriptability. Many integrated development environments follow this pattern, in which the graphical user interface invokes the compiler, linker, and other tools using command-line interfaces. Particularly relevant is the Commando dialog box system in the Macintosh Programmer's Workshop [1], which allows a developer to specify a dialog box interface for an arbitrary Macintosh command-line program. A Commando dialog box resource is an abstract description specifying the dialog box controls and how the controls are mapped to command-line options. In that sense, it resembles an HTML interface, but is more platform-dependent than HTML.

Others have investigated wrapping HTML interfaces around command-line programs on a web *server*, but not on the client. For example, Phanouriou and Abrams [19] described an HTML interface that presented status information about a web server (network, filesystem, memory, kernel, etc.) obtained from Unix commands.

The browser-shell is not the first alternative to the stan-

dard typescript Unix shell. Another is Sam [21], a graphical text editor which integrates external program execution in three ways: "< *command*" replaces the current selection with the output of a command, "> *command*" runs the command with the current selection as input, and "| *command*" redirects both input and output. The Emacs *shell-command-on-region* command provides similar capabilities. In a later editor, Acme [20], each external command's output appears in a new window, with a *tag line* similar to a browser's Location box that can be used to invoke another external program. Unlike Sam, Acme had no provision for supplying a command's input from a window, and both systems lacked the output history provided by a browser-shell.

### 3 User Interface

We now describe some important features of the browser-shell user interface. The first section is a summary of some previous work on which we are building. Subsequent sections describe new work: the command interpreter, web automation, creating web scripts by example, and invoking external programs and CGI programs.

#### 3.1 LAPIS

The web browser we used to prototype the browser-shell is called LAPIS (Figure 1), part of a system of generic tools for structured text that we call *lightweight structured text processing* [14]. Lightweight structured text processing enables users to define text structure interactively and incrementally, so that generic tools can operate on the text in structured fashion. Our lightweight structured text processing system has four components:

- a *pattern language* for describing text structure;
- *parsers* for standard structure, such as HTML and programming language syntax;
- *tools* for manipulating text using structure, including sorting, searching, extracting, reformatting, editing, computing statistics, graphing, etc;
- a *document viewer* (in this case, a web browser) for viewing documents, developing and testing patterns, and invoking tools.

LAPIS includes a new pattern language called *text constraints*. Text constraints describe a set of regions in a page in terms of relational operators,

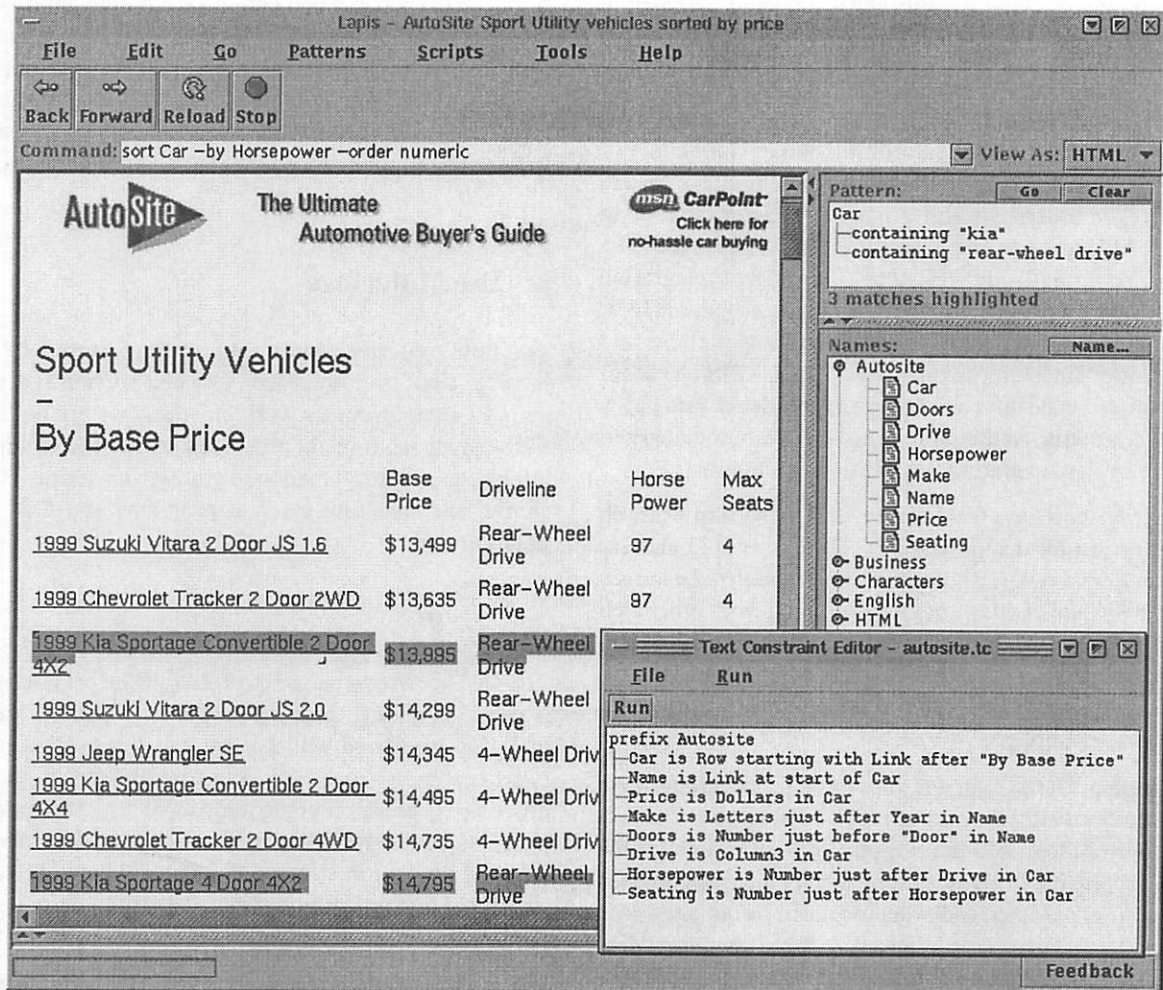


Figure 1: The LAPIS web browser, displaying a web page that lists new cars. The page structure is described by patterns shown in the inset window (Text Constraint Editor). Some of the terms used in these patterns (Row, Link, etc.) are defined by other patterns not shown, and others are defined by the built-in HTML parser. The user has entered a pattern in the Find box to highlight certain cars (rear-wheel drive Kias), and is now about to run a command in the Command box to sort all cars by horsepower.

such as *before*, *after*, *in*, and *contains*. Text constraints can refer to structure defined by arbitrary parsers, such as the built-in HTML parser that identifies HTML elements and assigns them names, such as Link, Paragraph, and Heading. A single text constraint pattern can refer to multiple parsers — for example, Line at start of Function refers to both Line, a name defined by a line-scanning parser, and Function, a name defined by a programming-language parser. In general, text constraints are designed to be more readable and comprehensible for users than context-free grammars or regular expressions, because a structure description can be reduced to a list of simple, intuitive constraints which can be read and understood individually. More details about the text constraints language can be found in a previous paper [14].

The LAPIS browser includes several tools for transforming web pages. For example, *keep* extracts a set of regions matching a text constraint pattern, *delete* deletes a set of regions, *sort* sorts a set of regions in-place, and *replace* replaces a set of regions with some replacement text. In the LAPIS browser described in a previous paper [14], a tool could only be invoked from a menu, and its output was directed to a new page in the browser. The browser-shell extensions described in this paper make it possible to invoke these tools from the Location box and from user-defined scripts.

### 3.2 The Browser-Shell

In order to create scripts of commands, we embedded Tcl [18] into LAPIS. Tcl was chosen partly because of its syntactic simplicity, and partly because a good Java implementation was available [5]. Tcl is also well-suited to interactive command execution.

Instead of presenting a Tcl interpreter in a separate window, LAPIS integrates the interpreter directly into the browser window. Tcl commands may be typed into the Location box. The typed command is applied to the current page, and its output is displayed in the browser as a new page that is added to the browsing history.

Using the Location box as a command line has several advantages. The page generated by a command can be browsed like a page generated by a URL. The browsing interface — Back, Forward, Stop, and Reload — also applies to command outputs. The Back button returns the browser to the previous page, Stop aborts a long-running command, and Reload runs the command again.

Since either a URL or a command can be typed into the Location box, LAPIS must be able to distinguish between them. The problem is trivial if the typed entry begins with a protocol prefix, such as `http:` or `file:`,

and LAPIS also recognizes the protocol `cmd:` for invoking a command unambiguously. If the typed entry does not begin with a prefix, LAPIS tries every possible interpretation: first as a command to execute, then as a filename to display, then as a domain name for a web server. This is an extension of the heuristics already used by the Location box of most web browsers.

For security reasons, LAPIS only executes a `cmd: URL` if it originates locally — e.g., if it is typed into the Location box or found in a page loaded from the local filesystem. A link in a remote web page cannot invoke a Tcl command.

### 3.3 Web Automation

Web browsing has two basic actions: clicking on hyperlinks and submitting forms. Automating web browsing requires equivalent script commands for these actions.

Clicking on a link has the same result as typing in its URL in the Location box. Thus the script command for clicking on a link is simply the link's URL, such as:

```
http://weather.yahoo.com/
```

For some links, however, the URL varies depending on when the page is viewed. Variable links are often found in online newspapers, for example, where links to top stories change from day to day. The `click` command can be used to click on a variable link by describing its location in the web page with a LAPIS text constraint pattern. For example:

```
http://www.salon.com/ # Start at Salon
click {Link after Image in Column3}
      # Click on top story
      # (curly braces are Tcl quoting)
```

For entering data into forms, the `enter` command is provided, with two arguments. The first argument is a pattern describing the form field to affect. Since HTML form fields are named, this pattern may simply be the field name. Alternatively, the pattern may describe the field in terms of its context (e.g., `TextBox` just after "Email Address:"), which has the advantage of being comprehensible without looking at the HTML source. The second argument to `enter` specifies the value to enter in the field. For text fields, this value is entered in the field directly. For menus or lists, the value is selected in the list. For radio buttons or checkboxes, the value should be "on" or "off" (or yes/no, true/false, or 0/1).

Forms are submitted either by a `click` command describing the form's submit button. For example, here is a complete script that searches Google for the USENIX 2000 conference home page:



```

http://www.google.com/
enter {Textbox just after \
      "Search the web using Google"} \
      {USENIX 2000}
click SubmitButton

```

LAPIS also provides script commands for other web browsing actions, including Home, Back, Forward, Stop, Reload, and Save.

The examples presented so far have been web-site-specific, but some browsing tasks are sufficiently uniform across web sites to be handled by a generic script. For example, the following script can log into many web sites, assuming the user's login name and password have been stored in the Tcl variables `id` and `password`:

```

enter {Textbox \
      just after Text containing \
      ("login"|"email"|"id"|"user")} \
      $id
enter {Textbox just after Text containing \
      "password"} \
      $password
click SubmitButton

```

### 3.4 Automation by Demonstration

To create a browsing script quickly, the user can *demonstrate* it by recording a browsing sequence. The demonstration begins with an arbitrary example page, the *input page*, showing in the browser. Invoking the Demonstrate command pops up a new browser window, in which the browsing demonstration will take place. A new window is created so that the browsing sequence can refer to the input page for parameters. Like any LAPIS browser window, the Demonstrate window records a browsing history: URLs visited and commands typed. Unlike a normal browser window, however, the Demonstrate window's history also records user events in form controls. For example, if the user types into a form field, the history will record an equivalent `enter` command.

To fill in a form with text from the input page, the user can make a selection in the input page, then drag-and-drop (or copy-and-paste) to a form field in the Demonstrate window. If the copied text was selected by searching for a pattern, then this action records the command `enter field-name pattern` in the history. If the copied data was selected manually, then the command `enter field-name {Selection}` is recorded in the history. When the script is run at a later time, `Selection` will return the user's selection at that time. More complex dependencies can be expressed by typing a Tcl command instead of pointing-and-clicking. For example, if a radio button should be selected only if the input page has certain features, then the user

might type the command `if {[find pattern]} {click field-name}`.

Using Back and Forward, the user can revise the demonstration as necessary until the desired results are achieved. The browsing history, which is essentially a Tcl script, can also be opened in an editing window, where the user can insert conditionals, iteration, and comments, if desired. When the user is satisfied with the demonstration, the Demonstrate window is closed, the history is saved as a script, and the script becomes available as a named command.

LAPIS demonstrations have two advantages over the macro recorders in previous systems, such as LiveAgent [11]. First, the recorded transcript is represented by the browsing history, which is visible, easy to navigate, and very familiar. A crucial part of making this work is that LAPIS inserts commands as well as URLs in the browsing history. Second, an experienced user can generalize the demonstration on-the-fly by typing commands at crucial points instead of pointing-and-clicking. Since a full scripting language is supported, the resulting scripts can be significantly more expressive than recorded macros, without taking much more time to develop.

### 3.5 Script Optimization

A script created by demonstration may include unnecessary steps, which may be expensive if they fetch web pages. To address this problem, LAPIS includes an optimizer that tries to compact the browsing script. For example, a sequence of simple link-clicking may result in a list of URLs:

```

# Start at Yahoo
http://www.yahoo.com/

# Click on Weather
http://weather.yahoo.com/

# Click on US
http://weather.yahoo.com/regional/US.html

```

Since the URLs are constant, depending neither on the input page nor on previous pages in the demonstration, the optimizer can delete all but the last, saving several page fetches.

The optimizer can also streamline form submissions. Submitting a form normally requires two page fetches, one to retrieve the form and another to submit the form. The optimizer can eliminate the first fetch by hard-coding the form submission URL, the form field names, and their values.



These optimizations are not always safe, however. For example, some forms have a variable submission URL or variable default values, often referring to unique session identifiers or persistent state. Thus the optimizer does not run by default. Instead, the user selects some or all of the script and invokes the optimizer on it manually. In the future, the optimizer may be able to gather information from repeated runs of a script to determine which optimizations would be safe to make automatically.

An optimized form submission may stop working correctly if the form changes, which happens from time to time when web sites are redesigned or moved. Gross changes can be detected by various techniques, such as the modification time or checksum of the form page, but the cost of detecting changes in just the *form* (as opposed to page content around the form, which might change often) would overwhelm the savings of optimization. This is a special case of a general challenge for web automation: recognizing and dealing with change on the Web. LAPIS helps with the problem by providing a rich pattern language, enabling browsing scripts to be insulated from many kinds of changes, but otherwise leaves detecting and debugging broken scripts to the user.

### 3.6 External Programs

In addition to built-in Tcl commands, LAPIS can also run an external command-line program from the Location box. If the command name is not found as a built-in Tcl command or user-defined script, then LAPIS searches for an external program by that name. If an external program happens to share the same name as a Tcl command, the user can force the external program to run with the `exec :` prefix.

Like a Tcl command, an external program is applied to the current page and displays its output as a new page added to the browser history. For example, if the user types (on BSD-style Unix) `ps aux`, then the browser displays a list of running processes. If the next command is `grep xclock`, then the process listing is filtered to display only those lines containing "xclock."

To make this work with legacy programs such as `ps` and `grep`, the external program is invoked in a subprocess with its input and output redirected. Standard input is read from the current page of the browser, passing the HTML source if the current page is a web page. Standard output is sent to a new page of the browser, which is displayed incrementally as the program writes output. Standard error is sent to a subframe of the page, to separate it from standard output.

A program's output may be parsed and manipulated like any other page in LAPIS. For example, `ps aux` displays information about running processes:

```
USER      PID %CPU %MEM    SIZE   RSS  TTY...
bin        160  0.0   0.4    752   320  ? ...
daemon    194  0.0   0.6    784   404  ? ...
rcm       294  0.0   1.0   1196   660  ? ...
```

The output of `ps` can be parsed by simple LAPIS text constraint patterns:

```
Process = Line,
    but not starting with "USER";
User = Alphanumeric at start of Process;
PID = Number just after User;
```

These identifiers can be used with LAPIS commands that search and manipulate the output of `ps`:

```
# sort processes by PID
sort Process -by PID -order numeric

# display only xterm processes
keep {Process containing "xterm"}

# kill all xterm processes
kill [extract {PID in Process \
    containing "xterm"}]
```

By default, patterns and commands are applied only to standard output, but standard error may also be processed by referring to the Tcl variable `$error`, as in `find {"Warning:"} $error`.

### 3.7 CGI Programs

If an external program outputs HTML instead of plain text, the browser-shell detects it and renders it as a web page. HTML output is detected by several simple heuristics, such as an initial `<html>` or `<doctype>` tag.

The HTML output may contain embedded forms. To submit a filled-out form back to the external program, LAPIS passes form parameters using the Common Gateway Interface (CGI) [17]. CGI passes form fields and other request information by setting environment variables, such as `QUERY_STRING`. Although CGI is commonly used by web servers to invoke external programs, no major web browser can invoke a CGI program locally. (The closest we've found is the Help Viewer in KDE 1.1, which displays HTML help documents and uses CGI to invoke a local search engine.) One beneficial side-effect of using CGI to communicate with external programs is that existing CGI scripts can be run directly by the browser-shell, without installing them in a web server. This feature may be useful for developing and testing CGI applications outside a web server.

Whether a form is being submitted or not, LAPIS always sets the CGI environment variables when it invokes an

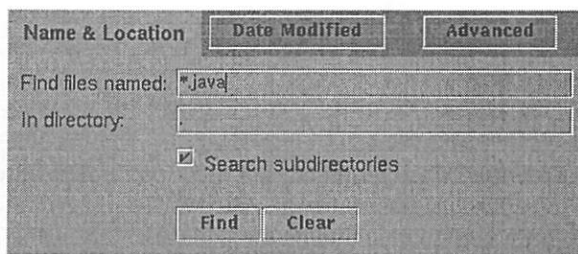


Figure 2: HTML interface for Unix *find*.

external program. A program can use the presence or absence of these variables to determine whether it was invoked from the browser-shell, in which case it can present an HTML interface and act like a CGI program, or from the ordinary typescript shell, in which case it should present a text-only or command-line interface.

One use for this facility is wrapping a friendlier HTML interface around an existing command-line program. For example, some users have trouble remembering the syntax for the Unix *find* command, which searches for files matching certain constraints. *Find* supports a variety of predicates on filename, date, user ownership, etc., and Boolean operators for combining predicates. We wrote a Perl CGI script wrapper around *find* which displays a simple HTML form (Figure 2). The first part of the wrapper script (Figure 3) tests whether the script is running under LAPIS. If not, or if the user passed command line arguments, then the wrapper simply invokes the original *find*. Otherwise, the script prints an HTML page containing the form. When the form is filled out and submitted back to the wrapper script, the script invokes *find* appropriately.

The HTML wrapper makes it possible to use *find* without learning or remembering its command-line syntax. A GUI frontend for *find* would offer the same benefits, but at greater cost: a GUI frontend has no ready hooks for automation, but the HTML form interface can be scripted in LAPIS exactly as if it were a web service. For example, a Java programmer may want a script that searches all subdirectories for files ending with *.class* and stores them in a ZIP file. The user pops up a Demonstrate window, invokes *find* to display its HTML form, fills in the form to search for files named *\*.class*, and applies *zip* to the resulting list of files. This sequence of actions is then saved as a script. Thus the user can include *find* in a script without learning its more complex command-line interface.

```
#!/usr/bin/perl -w

# Check if invoked outside of browser-shell
# or passed arguments.
if (!defined $ENV{"GATEWAY-INTERFACE"})
    || @ARGV > 0 {
    # Pass arguments directly to find
    exec ("/usr/bin/find", @ARGV);
}

# Otherwise act as CGI script.
use CGI qw/:standard/;

if (!param()) {
    # No form submitted.
    # Display the HTML interface.
    exec "cat /usr/doc/find/find-form.html";
} else {
    # Handle form submission.
    exec ("/usr/bin/find",
        param("directory"),
        param("search.subdirectories")
        ? () : ("-maxdepth", "1"),
        "-name", param("name"),
        "-print");
}
```

Figure 3: Perl wrapper for *find* that displays the HTML form interface shown in Figure 2 when invoked inside the browser-shell. Form submissions are handled by the Perl CGI.pm module.

## 4 Implementation

The browser-shell prototype described in this paper was implemented by modifying an existing web browser, LAPIS, originally designed to test new user interface ideas. LAPIS is written in Java 1.1 using the HTML layout component JEditorPane from the Java Foundation Classes. Before modification, LAPIS consisted of about 18,000 lines of code. The browser-shell features added about 2,000 lines of code. The LAPIS browser-shell has been tested on Linux, Solaris, and Windows NT.

Modifying a browser is not the only way to implement browser-shell capabilities. Two other general strategies exist for adding features to web browsers. One scheme uses an internal browser extension mechanism, such as a Netscape plugin, an Internet Explorer ActiveX component, or a Java applet. The other scheme is an HTTP proxy, external to the browser but running on the same machine, that filters the browser's HTTP requests.

Both schemes have the advantage of working with existing browsers, but lack of tight integration with the browser makes some browser-shell features difficult or impossible to implement. For example, neither scheme would allow commands to be typed directly into the browser's Location box. Highlighting the results of pattern matches would be much harder, as would monitoring the user's entries in form fields to generate scripts by

demonstration. The lack of control over the browser's user interface makes these browser-extension schemes too constraining for use as a research testbed. For a developed product, however, one of these schemes may be the best bet, even if it can only deliver a subset of the capabilities described in this paper.

We suggest that there are several levels of browser-shell complexity. Higher levels are harder to design and implement, but deliver correspondingly greater benefits. In increasing order of complexity, the levels are:

1. *Local program invocation.* Implementing this level requires spawning a subprocess and redirecting its input and output to the browser. This level is sufficient for using the browser as a command shell.
2. *Local CGI invocation.* Implementing this level requires encoding a form submission into environment variables and invoking a local program. This level is sufficient to support local HTML interfaces with form submission.
3. *Embedded scripting language.* Many web browsers already embed Javascript, but do not support automatic browsing (i.e., a sequence of script commands invoked on successive web pages). With automatic browsing, this level is sufficient to support web automation.
4. *Embedded pattern language.* A pattern language like text constraints enables the user to describe, manipulate, and extract parts of web pages and program outputs. This level acts as a glue language for connecting unrelated information sources or programs, so an ideal pattern language should be capable of describing not only HTML, but also text and XML.
5. *Web automation by demonstration.* Implementing this level requires recording user events and generalizing them into script commands. This level helps novice users learn the scripting language and helps expert users streamline the construction of scripts.

## 5 Discussion

We now discuss some general implications of integrating a command shell into a web browser, in particular the new applications, architectures, and interaction styles that such a hybrid enables.

### 5.1 HTML Interfaces

Much interest in recent years has focused on creating and deploying HTML-based applications that run in web servers. The advantages of deploying an application as a web service are well understood: it can be accessed by millions of users at the click of a button, it can be upgraded easily, and it can even be given away for free, paid for by advertising. The most popular sites on the Web are HTML interfaces in this sense.

The browser-shell opens up a new possibility: deploying HTML interfaces on the client. There are still many reasons to deploy applications on the client, including performance, security, and ability to run disconnected from the network. Current browsers cannot submit HTML forms to client-side programs, however, forcing a client-side HTML application to handle its user input in a more complicated way (e.g., with Javascript, Java, or ActiveX). The browser-shell's ability to submit forms to local programs allows client-side programs to have pure HTML user interfaces, displayed entirely in the browser.

HTML interfaces have several advantages. First, an HTML interface is easy to implement portably, since it needs only the standard I/O library rather than larger, less portable GUI libraries. Second, a wide variety of HTML editors and CGI libraries already exist, making the job easier. Third, compared to a command-line interface, an HTML interface is easy to use, not only because it is visual, but also because users are familiar with similar interfaces on the Web. Finally, compared to a GUI, an HTML interface is easier to script because it is declarative and textual, allowing systems like LAPIS to parse the interface and control it automatically.

Some applications are well-suited to HTML; others are not. User input is limited to forms with standard controls such as buttons, menus, and text fields, so applications that demand richer interaction would be poorly suited. On the other hand, applications with high information content, such as detailed help or reference materials, would be well-suited, since HTML makes it easy to intersperse forms with formatted text, pictures, and hyperlinks.

Any program that already has a command-line interface is a prime candidate for an HTML interface. As our *find* example showed, wrapping an HTML interface around a legacy program is simple if the program takes all user input as command-line arguments. Programs that conduct an interactive dialog with the user are trickier to wrap, however, because the CGI protocol does not support persistent connections. The wrapper must be reinvoked for every form submission. This problem could be solved by a more complex wrapper that maintains its own persistent connection to the legacy program, or by



an alternative form submission protocol with a persistent connection to the wrapper.

HTML interfaces allow command-line programs to be self-describing. Instead of the terse “usage” message printed by command-line interfaces, a program running in a browser-shell would display its HTML documentation, and embedded in the documentation itself would be the program’s user interface. Thus, the usage message of an HTML interface not only explains what the program does, but also presents an interface for actually invoking it.

## 5.2 New Shell Interaction Model

The web browser is becoming a central part of the desktop interface. Modern browsers, such as Microsoft Internet Explorer and KDE’s *kfm* [10], already include file management among the web browser’s responsibilities. Integrating the system command prompt is another step along the same path, which makes sense because file management and command execution are often intertwined.

The browser-shell interface behaves differently from a traditional typescript shell, however. Whereas a typescript shell interleaves commands with program output in the same window, a browser-shell separates the command prompt from program output. The browser-shell also automatically redirects program input from the current browser page, and automatically sends program output to a new browser page.

One effect of these differences is on scrolling. In a typescript interface, long output may scroll out of the window. To view the start of the output, the user must either scroll back, or else rerun the command with output redirected to *more* or *head*. The browser-shell, by contrast, initially displays the *first* windowful of output, rather than the *last*, reducing the need for scrolling. When output is less than a windowful, a typescript can become cluttered by outputs of several commands, forcing the user to scan for the start of the latest output. The browser-shell displays each program output on a new, blank page. The overall effect of the browser-shell is like automatically redirecting output to *more*.

Unlike *more*, however, the browser-shell’s display is not ephemeral. The displayed output can be passed as input to another command, which allows pipelines to be assembled more fluidly than in the typescript interface. Developing a complicated pipeline, such as `ps ax | grep xclock | cut -d ' ' -f 1`, is often an incremental process. In typescript interfaces, where input redirection must be specified explicitly, this process typically takes one of two forms:

- Repeated execution: run A and view the output; then run A | B and view the output; then (B turned out wrong) run A | B’ and view the output; etc. This strategy fails if any of the commands run slowly or have side-effects.
- Temporary files: run A > t1 and examine t1; then run B < t1 > t2 and examine t2; then (B was wrong) run B’ < t1 > t2, etc.

The browser-shell offers a third alternative: run A and view the output; then run B (which automatically receives its input from A) and view the output; then press Back (because B was wrong) and run B’ instead. The browser-shell displays each intermediate result of the pipeline while serving as automatic temporary storage.

Automatic input redirection makes constructing a pipeline very fluid, but it is inappropriate for programs that use standard input for interacting with the user, such as *passwd*. Such programs cannot be run in a browser-shell without modification, such as wrapping an HTML interface around the program, or running the program in a terminal emulator, possibly embedded in the browser-shell window.

One problem with the browser-shell model is the linear nature of the browsing history. If the user runs A, backs up, and then runs B, the output of A disappears from the browsing history. To solve this problem, the LAPIS prototype lets the user duplicate the browser window, including its history, so that one window preserves the original history while the other is used to backtrack. (Netscape’s New Window command worked similarly before version 4.0.) A more complex solution might extend the linear browsing history to a branching tree [2].

## 6 Status and Future Work

The LAPIS web browser described in this paper, including Java source code, is available from

<http://www.cs.cmu.edu/~rcm/lapis/>

LAPIS is only a prototype, but it demonstrates the basic ideas described in this paper. Unfortunately, the LAPIS prototype is not robust enough for everyday use, largely because JEditorPane renders many web pages poorly. An important avenue of future work will be to convert a production-quality web browser into a browser-shell and experiment with using it on a daily basis.



Several features are needed to make the browser-shell more useful and more efficient as a command shell, including:

- Background processes. Web browsers generally stop loading a page when a new URL is typed in the Location box. Similarly, LAPIS automatically stops the currently executing command when a new command is typed. As a result, only one command can be running in each LAPIS browser window. An improvement would be support for background-process syntax. If a command ends with `&`, it could continue running in the background, storing its output in case the user ever backs up through the history.
- Handling large outputs. A command may generate too much output for the browser to display efficiently. The same problem often happens in type-script shells, usually forcing the user to abort the program and run it again redirected to a file. To handle this problem, the browser could automatically truncate the display if the output exceeds a certain user-configurable length. The remaining output would still be spooled to the browser cache, so that the entire output can be viewed in full if desired, or passed as input to another program.
- Streaming I/O. A pipeline may process too much data for the browser's limited cache to store efficiently. Although the browser-shell's automatic I/O redirection could still be used to assemble the pipeline (presumably on a subset of the data), the pipeline would run better on the real data if its constituent commands were invoked in parallel with minimal buffering of intermediate results. The browser-shell might do this automatically when invoking a script.
- Shell syntax. Expert users would be more comfortable in the browser-shell if it also supported conventional operators for pipelining and I/O redirection, such as `|`, `<`, `>`, and `>>`. The most direct way to accommodate expert users might be to embed an existing shell, such as *bash* or *tcsh*, as an alternative to *Tcl*.

## 7 Conclusions

We have integrated a command shell into a web browser, and shown how this arrangement delivers benefits in three areas: (1) web automation; (2) HTML user interfaces for command-line applications; and (3) using a

web browser as a new way to interact with the system command prompt.

We would hope that the next generation of web browsers will include at least some of these features, enabling future web users to put the power of automation to work in browsing and manipulating the Web.

## Acknowledgements

The authors are grateful to David Garlan, John Pane, and the anonymous referees for their helpful suggestions on improving this paper.

This research was funded in part by a USENIX Student Research Grant.

## References

- [1] Apple Computer, Inc. *Macintosh Programmer's Workshop*. <http://devworld.apple.com/tools/mpw-tools/>
- [2] E.Z. Ayers and J.T. Stasko. "Using Graphic History in Browsing the World Wide Web." *Proc. 4th International World Wide Web Conference WWW4*, December 1995, pp 259–270.
- [3] K. Borg. "IShell: A Visual UNIX Shell." *Proc. Conference on Human Factors in Computing Systems (CHI '90)*, 1990, pp 201–207.
- [4] M. A. Cusumano and D. B. Yoffie. "What Netscape Learned From Cross-Platform Software Development." *Comm. ACM*, v42 n10, October 1999, pp 72–78.
- [5] M. DeJong, et al. *Jacl and Tcl Blend*. <http://www.scriptics.com/software/java>
- [6] P. E. Haeberli. "ConMan: A Visual Programming Language for Interactive Graphics." *Proc. ACM SIGGRAPH 98*, 1988, pp 103–111.
- [7] T. R. Henry and S. E. Hudson. "Squish: A Graphical Shell for Unix." *Graphics Interface*, 1988, pp 43–49.
- [8] B. Jovanovic and J. D. Foley. "A Simple Graphics Interface to UNIX." Technical Report GWU-IIST-86-23, George Washington University Institute for Information Science and Technology, 1986.
- [9] T. Kistler and H. Marais. "WebL - A Programming Language For the Web." In *Computer Networks and ISDN Systems (Proc. 7th International World*

- Wide Web Conference WWW7), v30, April 1998, pp 259–270. Also appeared as DEC SRC Technical Note 1997-029.
- [10] K Desktop Environment. *KFM*. <http://www.kde.org/>
  - [11] B. Krulwich. “Automating the Internet: Agents as User Surrogates.” *IEEE Internet Computing*, v1 n4, July/August 1997. <http://computer.org/internet/v1n4/krul9707.htm>
  - [12] R. C. Miller and K. Bharat. “SPHINX: A Framework for Creating Personal, Site-Specific Web Crawlers.” In *Computer Networks and ISDN Systems (Proc. 7th International World Wide Web Conference WWW7)*, v30, April 1998.
  - [13] R. C. Miller and Brad A. Myers. “Creating Dynamic World Wide Web Pages By Demonstration.” Carnegie Mellon University School of Computer Science Tech Report CMU-CS-97-131 (and CMU-HCII-97-101), May 1997.
  - [14] R. C. Miller and B. A. Myers. “Lightweight Structured Text Processing.” *Proc. USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999, pp 131–144.
  - [15] F. Modugno and B. A. Myers. “Typed Output and Programming in the Interface.” Carnegie Mellon University School of Computer Science Technical Report, no. CMU-CS-93-134. March 1993.
  - [16] F. Modugno and B. A. Myers. “Pursuit: Visual Programming in a Visual Domain.” Carnegie Mellon University School of Computer Science Technical Report, no. CMU-CS-94-109. January 1994.
  - [17] NCSA. *Common Gateway Interface*. <http://hoohoo.ncsa.uiuc.edu/cgi/>
  - [18] J. Ousterhout. “Tcl: An Embeddable Command Language.” *Proc. USENIX 1990 Winter Technical Conference*, pp 133–146.
  - [19] C. Phanouriou and M. Abrams. “Transforming Command-Line Driven Systems to Web Applications.” *Proc. 6th International World Wide Web Conference (WWW6)*, 1997, Santa Clara CA, pp 599–606.
  - [20] R. Pike. “Acme: A User Interface for Programmers.” *Proc. USENIX 1994 Winter Technical Conference*.
  - [21] R. Pike. “The Text Editor sam.” *Software Practice & Experience*, v17 n11, November 1987, pp 813–845.
  - [22] A. Sugiura and Y. Koseki. “Internet Scrapbook: Automating Web Browsing Tasks by Demonstration.” *Proc. ACM Symposium on User Interface Software and Technology (UIST 98)*, 1998, pp 9–18.

# Operating System Support for Multi-User, Remote, Graphical Interaction

Alexander Ya-li Wong, Margo Seltzer

Harvard University, Division of Engineering and Applied Sciences

aywong@aywong.com, margo@eecs.harvard.edu

## Abstract

The emergence of thin client computing and multi-user, remote, graphical interaction revives a range of operating system research issues long dormant, and introduces new directions as well. This paper investigates the effect of operating system design and implementation on the performance of thin client service and interactive applications. We contend that the key performance metric for this type of system and its applications is user-perceived latency and we give a structured approach for investigating operating system design with this criterion in mind. In particular, we apply our approach to a quantitative comparison and analysis of Windows NT, Terminal Server Edition (TSE), and Linux with the X Windows System, two popular implementations of thin client service.

We find that the processor and memory scheduling algorithms in both operating systems are not tuned for thin client service. Under heavy CPU and memory load, we observed user-perceived latencies up to 100 times beyond the threshold of human perception. Even in the idle state, these systems induce unnecessary latency. TSE performs particularly poorly despite scheduler modifications to improve interactive responsiveness. We also show that TSE's network protocol outperforms X by up to six times, and also makes use of a bitmap cache which is essential for handling dynamic elements of modern user interfaces and can reduce network load in these cases by up to 2000%.

## 1 Introduction

Continuing improvements in networking speed, cost, standardization, and ubiquity have enabled a literal "explosion" of the traditional computer system architecture. Network links are now replacing an increasing number of internal system busses, allowing the processor, memory, disk, and display subsystems to be spatially extruded throughout a network. Well-known examples of extruded subsystems include distributed shared memory and network file systems [17, 12]. While these are established areas of research, much less of the literature addresses systems that enable extrusion of the display and input

subsystems that interface with human users.

Such functionality is commonly referred to as *thin client* computing. Driven by IT concerns over cost and manageability, the thin client trend has triggered renewed interest in X Windows-like schemes and the introduction of thin client service into major commercial operating systems. This trend will accelerate as consumer products such as personal digital assistants, cellular phones, pagers, and hand-held e-mail devices evolve and converge into a yet another class of thin client terminals, these additionally being wireless, mobile, and ubiquitous.

We explore the question of how, in the current revival of interactive timesharing, underlying operating system design impacts thin client service, and how current metrics are inadequate for capturing the relevant impacts. The contributions of this paper are an approach for analyzing the performance and scalability of thin client server operating systems, a quantitative comparison and analysis of Windows NT, Terminal Server Edition and Linux with X Windows, two popular implementations, and a discussion of related work informed by our approach.

The balance of this paper is organized as follows. Section 2 gives background on the X Windows System and Windows NT, Terminal Server Edition. Section 3 describes our approach. Sections 4, 5, and 6 discuss the processor, memory, and network in turn as resources within our framework. In Section 7, we discuss related work on thin client performance, and we conclude in Section 8.

## 2 TSE and Linux/X Windows

For the remainder of this paper, we will use conventional client/server terminology although it is backwards from the X Windows use. In particular, we will refer to the machine in front of the user as the client (although in X Windows the X server runs on this machine) and the machine on which the application runs the server (although X Windows clients, such as *xterm*, run on this machine).

TSE and X Windows share similar architectures. Applications running on the server request display updates and receive inputs from OS-provided abstractions. In X

Windows, this is the Xlib GUI library, and in TSE, the Win32 GUI library. The code underlying these libraries transparently routes display requests and input events either to local hardware or remote client devices over the network. Xlib interaction is entirely user-level, while TSE display requests pass through the kernel.

Multi-user capabilities in TSE are provided by recent modifications to the NT kernel, including virtualization of kernel objects across user sessions, per-session mappings of kernel address space, and code sharing across sessions. Multi-user functionality under X Windows is provided by the underlying Unix implementation. In this paper, we evaluate the performance of X Windows as it runs on top of the 2.0.36 Linux kernel.

The network protocols used to carry display and input information are X for X Windows and the Remote Display Protocol (RDP) for TSE. Both are high-level protocols encoding graphics primitive operations like bitblts and text, polygon, and line drawing.

RDP is implemented in TSE as a display device driver, and so appears to user-level applications as would any other display device, such as a graphics card. RDP also employs compression and client-side bitmap and glyph (text) caching to reduce network traffic. Varying levels of encryption can be applied to the protocol stream over the wire. TSE users run the TSE client software to connect to a TSE server. The client is a 200KB executable which is available for DOS, all Windows platforms, and Unix platforms via third-party add-ons. The RDP specification is not published by Microsoft, making some analyses more difficult. Although it is not within the scope of this paper, the reverse-engineering of RDP is part of our ongoing work.

X Windows is implemented on the server-side as a GUI library called Xlib that is linked into applications designed for the X environment. Such applications communicate over the wire directly and require no special support from the underlying operating system other than a TCP/IP stack. Client machines run so-called "X servers," which accept connections from X-enabled applications and display their interfaces. X server software is available on most major platforms including Windows, Unixes, and Macintosh.

We also include in our comparisons LBX, which is a protocol extension to X and is implemented as a proxy that lives on both ends of an X Windows connection. It takes normal X traffic and applies various compression techniques to reduce the bandwidth usage of X applications [9]. RDP, X, and LBX all ran over TCP/IP in our experimentation.

## 3 Our Approach

### 3.1 The Motivation

Traditional performance metrics in the systems domain do not apply to operating systems whose primary function is thin client service. The output of benchmark suites like Winstone, *Imbench*, and TPC are not particularly enlightening. Ultimately, those interested in deploying interface services need to know the maximum number of concurrent users their servers can support given some hardware configuration and what impact on users yields this maximum value. Each characteristic of thin client service places unique demands on what an appropriate benchmark must measure:

#### 3.1.1 Interactive

Unlike HTTP or database servers for which throughput is critical and response time often reduces to a question of throughput, the primary service in which we are interested is interactive login. Endo et al., argued persuasively that for desktop operating systems like Windows, latency, not throughput, is a key performance criterion [7]. We contend that for thin client servers, latency is not only a key criterion, it is paramount. Any useful metric must yield information on whether the system satisfies the latency demands of users.

#### 3.1.2 Multi-User

Benchmarks designed for single-user operating systems are not appropriate because a single user multi-tasking is not equivalent to multiple users uni-tasking or multi-tasking. On single-user systems, although asynchronous background tasks may consume system resources, system load is still typically limited by the rate at which the human user interacts with the foreground application. Furthermore, on a multi-user system there can be many foreground applications (one for each user), so latency demands must be met by more than just a single process. Furthermore, schemes to improve foreground application performance by favoring them over background processes are thwarted when most, if not all, processes are foreground.

#### 3.1.3 Graphical

In a graphical environment, the user's primary interaction is visual. Therefore, benchmarks need to consider the delicate tolerances of human perception, particularly with respect to latency.



### 3.1.4 Remote Access

On single-user systems like Windows 98 and NT Workstation, user interface richness and sophistication consume and are constrained by locally available video subsystem bandwidth. In remote-access environments like TSE and X Windows, the video subsystem at the server is irrelevant and the GUI is instead constrained by network bandwidth, the efficiency of the network protocol, and the video hardware at the client.

## 3.2 The Key Role of Latency

As discussed above, latency, not throughput, is the key performance criterion for thin client service. A user interacting with an operating system performs a set of operations by sending input and waiting for the system to respond. These are the operations on which the user is sensitive to latency. Previous work has found that tolerable levels of latency vary with the nature of the operation. For example, latency tolerances for continuous operations are lower than for discrete operations, and humans are generally irritated by latencies 100ms or greater [4, 13, 20]. Jitter, or an inconsistent level of latency, is also considered harmful.

The quality of a system can therefore degrade in three ways with respect to latency. First, for a given operation, latency can rise above perceptible levels, and performance suffers as latency continues to increase for that operation. Second, performance suffers as the number of operations that induce perceptible latency increases. Finally, performance suffers when perceptible latency continually changes and is unpredictable. Ideally, a “good” system would meet users’ latency demands for each operation performed and would do so consistently.

In an interface service environment, latency depends on three categories of factors:

### 3.2.1 Hardware resources

Relevant hardware resources include the processor, memory, disk, and network. Hardware inevitably introduces latency, and slow hardware contributes more. Latency can also be caused by resource scarcity. For example, while free memory remains, data access latency is bounded by the speed of the memory hierarchy level into which the active data set fits. But when physical memory is exhausted and paging to disk begins, average data access latency increases dramatically.

### 3.2.2 Operating system structure

Operating system design and implementation also influence user-perceived latency. Even bleeding-edge hard-

ware can be sabotaged if it is exposed to users through a poorly designed operating system abstraction. Long input handling code paths, inefficient context switches, bad scheduling decisions, and poor management of resource contention can all contribute to increased latency.

### 3.2.3 User behavior

User behavior indirectly affects latency through hardware resource limitations. Two classes of users running different application mixes will consume resources at different per-user rates. As concurrent use increases, the class of users with greater per-user resource demands will approach saturation conditions and potential increases in latency more quickly.

## 3.3 Applying our Approach

In the next three sections, we use these principles to guide a quantitative comparison of TSE and Linux/X Windows. Our analysis cuts along the axis of hardware resources, as we consider the processor, memory and the network in turn. For each resource, we consider the impact of user behavior and how the exercise of various applications generates load. Finally, we consider how load translates into user-perceptible latency, and how that translation is influenced by design characteristics of the operating system. Using this analysis, we highlight shortcomings in current operating system design with respect to thin client service, and suggest new potential directions for operating systems optimization. Some issues, such as network graphics and protocol efficiency, are unique to remote interactive access, while others, such as process and memory scheduling, have not been visited in the literature in quite some time but are critical in establishing a high-performance, thin client computing environment.

## 3.4 Experimental Testbed

Our testbed consisted of a server, a client, and a network listening host connected by a Bay Networks NetGear EN104TP 10Mbps Ethernet hub. The server was a 333MHz Intel Celeron system with the Intel 440EX AGPset, 96MB SDRAM, a 4GB IDE IBM DCAA-34330 hard disk, and a Bay Networks NetGear FA-310 Ethernet adapter. The client was an Intel Pentium II-400 with the Intel 440BX AGPset, 128MB SDRAM, an 11GB IDE Maxtor 91152D8 hard disk, and a 3Com 3C905B Ethernet adapter. The listening host was an Intel Pentium-233 with the Intel 440TX PCIset, 96MB EDO RAM, a 2GB IDE IBM DTNA-22160 hard disk, and a 3Com 3C589C Ethernet adapter. The server ran, alternately, the Linux 2.0.36 kernel and build 419 of Windows NT, Terminal

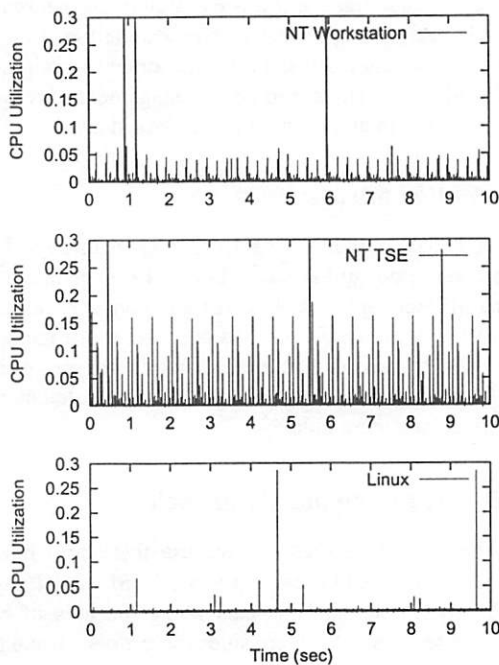


Figure 1: A comparison of the idle-state processor activity over a 10 second period in NT Workstation, TSE, and Linux.

Server Edition. The client system ran Windows 98 as its operating system and established remote access to the server with build 419 of the TSE client and White Pine's eXodus X-Windows server version 5.6.4. The listening host ran the 2.0.36 Linux kernel. All network adapters were configured for 10Mbps half-duplex operation.

## 4 Processor

In this section, we examine the latency characteristics of operating system processor abstractions in the context of thin client service. We consider how user behavior generates processor load and then compare each operating system's ability to minimize the user-perceived latency induced by varying levels of load.

### 4.1 From Behavior to Load

The addition of thin client support to an operating system has a measurable effect on both its requirements and characteristics with respect to latency.

When a system becomes multi-user (as NT did when Microsoft introduced TSE), it bears the additional burden of meeting human latency requirements for multiple concurrent foreground applications versus just one for single-user operation.

Support for multiple concurrent users produces additional system activity and potential latency increases for user-level applications. Multi-user support typically includes at least one daemon process to listen for and handle incoming session connections and additional per-user kernel state and ownership information. Remote-access support contributes yet more latency. Interface operations previously handled by just the graphics subsystem must now also pass through the network subsystem.

#### 4.1.1 Compulsory Load

These latency contributors, those that are inherent in the operating system, are particularly important because they are behavior-independent. Because multi-user and remote-access support are core functionality required for thin client service, all users, regardless of the applications they use, are subject to at least the minimum latency induced by these components. Any calculation of user-perceived latency must start by measuring this baseline load, which we call "compulsory load."

#### Methodology

Endo et al. introduced a novel methodology for measuring user-perceived latency they describe as "measuring lost time" [7]. Using a combination of the Pentium Performance Counters and system idle loop instrumentation, they are able to determine when, and for how long the CPU is busy handling user input events. This yields a method for measuring user-perceived latency with a precision not previously achieved.

Endo et al. employed this technique to generate a set of idle system profiles for three versions of Windows: 95, NT 3.51, and NT 4.0. We use this data as a baseline and use identical methodology to measure the idle system profiles of TSE and Linux for comparison. We discuss the significance of these idle-state profiles as it bears on latency in section 4.2.1.

#### From Windows NT to TSE

As shown in Figure 1, TSE exhibits greater overall idle-state CPU activity than NT does, demonstrating the increase in compulsory latency caused by the changes made to transform the NT kernel into the TSE kernel. Although Microsoft documentation states that the typical clock interval for NT 4.0 running on a Pentium processor is 15ms, we found, as Endo et al. did, small regular CPU spikes at 10ms intervals in both TSE and NT, suggesting that the clock interrupts are handled every 10ms [5]. This discrepancy is unexplained.

Beyond clock interrupt handling, TSE seems to perform a number of other activities at regular intervals that NT does not. These can be attributed to the addition of the Terminal Service and Session Manager which

listen for and handle incoming client connections, and whatever additional overhead there is in the idle-state for per-session state management in the NT Virtual Memory, Object, and Process Managers.

### X Windows on Linux

Unlike TSE, Unix has long had multi-user and local and remote graphical display capabilities courtesy of the X Windows System. Unix operating systems can also run in single-user mode, like NT, but seldom do, and usually only for the purposes of crash diagnosis and recovery.

Figure 1 also shows idle-state CPU activity for the Linux kernel running in multi-user mode. Clearly, the Linux kernel spends much less CPU time handling tasks when idle than do either NT or TSE. This contributes less compulsory load, which, as we will see in the next section, translates to less latency.

## 4.2 From Load to Latency

Next we discuss how system load translates to user-perceived latency. We examine how both compulsory and dynamic load can increase latency and how intelligent operating system design decisions can minimize those impacts.

### 4.2.1 Compulsory Latency

Figure 2 compares the cumulative latency of the three systems (NT, TSE, and Linux) in the idle state. The bulk of CPU activity under NT is attributable to events that are 100ms or shorter in duration. The TSE idle state sees these same events, plus a number of additional events lasting 250ms and 400ms. Linux, contrastingly, sees few idle events of significant latency. In the aggregate, TSE generates about three times the idle-state load that NT Workstation does, and about seven times that of Linux.

Even when the systems are idle, any user input activity that intersects with these events will experience delay. The scheduler design determines just how much delay there must be. In particular, the quantum (NT parlance for time-slice) is often manipulated to adjust the responsiveness of a system, but we find that the choice of quantum length is something of a “latency catch-22.”

Consider a round-robin scheduler and several non-blocking, ready-to-run threads with equal priority. The longer the quantum, the longer some thread three or four deep in the queue will have to wait until it can run. In contrast, if the quantum is made shorter, this inter-quantum waiting is reduced, but the full, run-to-block execution time of each thread becomes fragmented across more distinct quanta. If there are a large number of threads in the ready queue, then this problem of execution fragmentation can easily overwhelm the benefits of

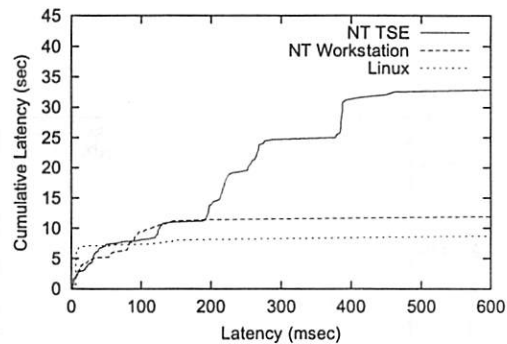


Figure 2: Idle-state profiles of the NT Workstation, TSE, and Linux kernels. The profiles are displayed as cumulative latency histograms. Large changes in the y-axis values indicate a significant contribution to overall latency by the events at the corresponding x-coordinate.

shorter quanta.

Consider a user operation that takes 100ms to complete with no other competing activity. Scheduler quanta on systems like NT and Linux are on the order of 10ms, meaning this operation is fragmented into as many as 10 quanta. If the system is busy with background processing, the original 100ms completion time can be extended considerably. While dynamic priority boosting for GUI-related and foreground threads may help alleviate this problem, it does not help when the competing threads are also foreground and/or GUI-related, as we would often find on a thin client server. Next, we discuss how effectively the TSE and Linux schedulers deal with this problem. For the following discussion, note that greater numeric priorities are better on NT and TSE, while lower are better on Unix systems.

### NT and TSE Scheduling

The NT and TSE kernels share the same scheduling code and differ only in their default priority assignments for processes and threads. NT Workstation and TSE both have a 30ms quantum on Intel Pentiums and higher. While TSE is based more directly on NT Server, which has a 180ms quantum, it uses the 30ms quantum found in NT Workstation, ostensibly to improve interactive responsiveness. The default priority level for foreground threads is 9 and for other threads it is 8.

The NT/TSE scheduler implements two mechanisms to further improve user interaction. The first, “quantum stretching,” allows the system administrator to multiply the quantum for all foreground threads. The allowed stretch factors are one, two, and three. The second mechanism is “priority boosting” for waiting GUI threads. GUI threads associated with an interactive ses-

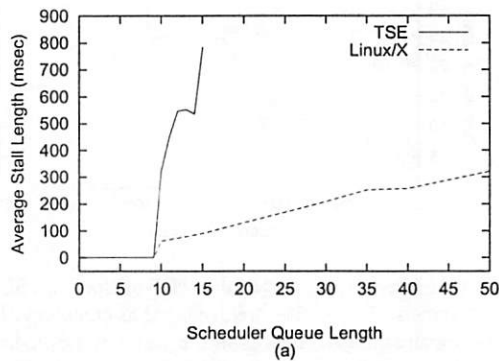


Figure 3: User-perceived keystroke handling latency as a function of server CPU load. Linux/X scales more gracefully than does TSE. TSE became unusable at a load of 15.

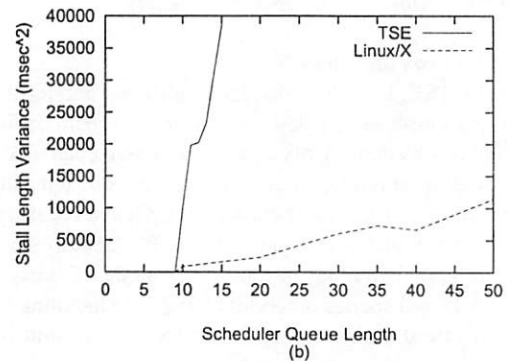


Figure 4: User-perceived keystroke handling jitter as a function of server CPU load. As with average latency, jitter is less noticeable on Linux/X than on TSE.

sion get their priority boosted to level 15 after waking up to service a user input event. This boost lasts for two quanta.

In TSE, the Session Manager and Terminal Service have a priority of 13. So, if the idle activity we saw in TSE is associated with these processes, GUI thread priority boosting should theoretically prevent this background activity from increasing user-visible latency since the boosted priority for the GUI thread is higher than that of the idle activity (15 vs. 13). However, the boost lasts for only two quanta, and even assuming they are stretched by three, this boost benefits the GUI thread for at most 180ms. After these 180ms, a GUI thread's priority drops back to 9, and cannot run again until all priority 13 threads yield or block. In the previous section, we saw idle-state events in TSE of up to 400ms. If a GUI thread intersects with such an event, its completion time would be extended by 400ms in spite of the scheduler's help.

When the system is dominated by foreground activity, a likely scenario on a thin client server hosting a large number of concurrent users, priority boosting provides no benefit for foreground threads with respect to one another. In this case, GUI threads can run for at most 90ms before being pre-empted by other threads in the queue.

The lesson here is that quantum stretching and priority boosting can only eliminate latency when the net boosted-priority "grace period" is long enough to complete the interactive operation. In the case of NT and TSE, this threshold is 180ms when competing with background activity, and 90ms when competing with other foreground activity.

How do these thresholds compare with typical interaction completion times? Endo et al. report that the majority of interactive events last between 30 to 45ms for Microsoft Notepad, 60 to 250ms for Microsoft Powerpoint, and 50 to 100ms for Microsoft Word. They also

report a 500ms completion time for the window maximize operation. We highlight this operation because it is representative of user interface elements meant to "enhance" the user experience, but are designed with fast, local, desktop systems in mind instead of thin client environments.

Note that Endo et al.'s values were collected on a 100Mhz Pentium machine, so these completion times are likely to be on the order of 5 to 10 times faster on today's processors. Nevertheless, continuing increases in user interface complexity, marked by more sophisticated graphics and the introduction of animated elements, makes such thresholds a continuing concern.

### Linux Scheduling

The Linux kernel supports FIFO, round robin, and other scheduling classes, with priority values between -20 and +20 in each class. Most processes run in the round robin class with a quantum of 10ms. There is no provision for changing the quantum length and no facility for automatic priority boosting on GUI-related or foreground processes. The first implication of this design is that any user input event that is greater than 10ms, which is a fairly low threshold, risks being fragmented across quanta. In a high load scenario, this can increase latency for that event considerably. And because the quantum is so small, the level of fragmentation is greater than in TSE.

Moreover, Linux provides no help for interactive processes. NT and TSE can easily target and boost foreground threads because of the tight integration of the graphics subsystem into the kernel. However, X Windows is a user-level graphics subsystem and there is no well-defined method for passing GUI-related information into the kernel.

But in 1993, Evans et al. of SunSoft did exactly that



when optimizing the System V, R4 scheduler for interactivity [8]. Their approach traded a clean user-level/kernel separation for scheduler access to application-level information on process interactivity. They showed that in a control system running the SVR4 kernel, keystroke handling latency increases as the scheduler queue length (or load) grows because there is no provision for protecting interactive processes from CPU-intensive processes. Then they demonstrated a prototype SVR4 kernel modified with an interactive scheduler for which keystroke handling latency remains constant and small, even as load approaches 20.

In spite of this work, years later no Unix-like kernels implement such improvements, probably due to the separation between kernel development and user-level X Windows development. Although the modifications made by Evans et al. should be readily adaptable to the Linux and BSD kernels, such an implementation faces several challenges. First, user-applications must be modified in order to take advantage of priority boosting. Kernel modifications and new system calls do not benefit existing X applications. Furthermore, because GUI services are provided by the user-level Xlib library and not the kernel, the kernel has no straightforward way of preventing arbitrary processes and threads from claiming they are GUI-related even when they are not. This challenge represents an opportunity for further research stemming from this work.

#### 4.2.2 Dynamic Latency

If scheduler deftness is important in minimizing compulsory latency, it is even more critical in the dynamic case when the processor is heavily loaded and many threads are queued, ready to run. Our experiments show that both TSE and Linux perform poorly when compared to Evans et al.'s modified SVR4, and surprisingly, TSE performs much more poorly than does Linux, despite its mechanisms for favoring interactive foreground processes.

#### Methodology

We wrote a simple C program called *sink* that is a greedy consumer of CPU cycles. Since *sink* never voluntarily yields the processor, each running instance should increase the scheduler queue length by one. We used this program to control the load level on the server.

At varying levels of load, we ran a simple text editing application at the client. Under TSE it was Notepad and under X Windows *vim*. The tester held down a key in the application to engage character repeat on the client machine, the rate of which was set at 20Hz.

Under no load, we expect the server to respond every 50ms with a screen update message to draw a new

character. We measured message inter-arrival times by examining the timestamps on the corresponding network packets using *tcpdump*. Under increasing load, we expect some or all of these inter-arrival times to rise above 50ms as the server handles other computations. We call each instance of this an "interactive stall," with the length of the stall defined as the inter-arrival time minus 50ms. A stall is therefore the duration of time when the server's processor was occupied with other tasks and the user therefore ended up waiting.

For each load level we recorded message inter-arrival times for 60 seconds and report the average.

#### Results

With no load on the server, both kernels performed as expected, sending a message to the client every 50ms. But figures 3 and 4 show that as the number of *sink* processes increased, so too did the length and variability of stalls. The relationship between load and latency exhibited by TSE and Linux are still similar to that of the 1993 vintage SVR4 scheduler.

TSE exhibits poor behavior, with latency increasing sharply around 10 load units. The data for TSE stops at 15 load units because at that point the system became barely usable at the console. These TSE results are inexplicable without access to NT source code. Linux, on the other hand, while still not protecting interactive processes properly, handles increasing load with more grace, with latency increasing linearly but more slowly with respect to load. This confirms the expectations based on our analysis of the scheduler.

These levels of latency and jitter are quite high and certainly well within the range of human perception. Subjectively, these interactive stalls felt like irregular "hiccups" in the system. That is, while some keystrokes would generate immediate character echo, the echo for other keystrokes was delayed by up to one second in extreme cases. The inconsistency of these stalls, or latency jitter, further contributed to an unsatisfying user experience.

## 5 Memory

In this section, we examine memory consumption and memory induced latency in a thin client context. First, we consider how varying user behavior consumes memory at different rates. Second, we consider the latency consequences of increasing memory usage on the server.

## 5.1 From Behavior to Load

### 5.1.1 Compulsory Load

Compulsory memory load has two components. The first is the dynamic memory usage of the kernel and the user-level services necessary to support graphical, multi-user, remote login. This is simply the amount of memory that is unavailable to user applications when the system is idle with no user sessions. In our configuration, memory load in this state was roughly comparable between the two systems, 17MB for Linux and 19MB for TSE.

The second component is the memory usage of each user session. This usage is governed by what is considered to be a minimal login with no additional user activity. The following tables list the processes associated with minimal logins for each system. For each process, we conservatively give the amount of private, per-user memory consumption, excluding any amortized shared code page costs for executable text or mapped, shareable memory.

Process	Typical
<i>in.rshd</i>	204 KB
<i>xterm</i>	372 KB
<i>bash</i>	176 KB
<i>Total</i>	752 KB

a. Linux/X

The *in.rshd* is a daemon process that accepts and processes client session initiation requests. It is somewhat analogous to the Terminal Service in TSE. *xterm* is the X-enabled application that draws a graphical terminal window to the client's screen. *bash* is the shell that gives the user access to operating system services and allows other processes to be launched.

Process	Typical	Light
(shell)	<i>explorer.exe</i> 1,368 KB	<i>command.com</i> 224 KB
<i>csrss.exe</i>	452 KB	452 KB
<i>nddeagnt.exe</i>	300 KB	300 KB
<i>winlogon.exe</i>	700 KB	700 KB
<i>Total</i>	2,820 KB	1,676 KB

b. NT TSE

*csrss.exe* is the subsystem that provides Win32 API services, *nddeagnt.exe* provides network dynamic data exchange (DDE) services, and *winlogon.exe* manages interactive user logons and logoffs.

In fairness, *explorer.exe* does offer a fully featured graphical file navigation and program launching interface while *bash* does not. We should note that TSE

clients have the option of dispensing with the Explorer and running just a specific application in a user session (but it cannot be empty). This admits the possibility of using a lighter alternative more comparable to *bash*. The DOS Prompt (*command.com*), for example, requires only 224KB of private, unshared memory, bringing the minimum compulsory memory load per-user for TSE down to 1,676KB.

### 5.1.2 Dynamic Load

Dynamic load is the memory utilization due to user behavior above and beyond the simple act of logging into the system. This issue is no different for thin client servers than for other types of operating systems. Assuming that the operating system supports code page sharing, the smaller the set of active applications and the smaller their user-specific stack and heap areas, the lower the dynamic memory load.

Of the three hardware resources discussed in this paper, memory is the most difficult for which to make generalizing comments regarding load. Memory utilization is highly dependent upon the applications used.

## 5.2 From Load to Latency

The latency consequences of increasing memory utilization are well-known. As the active data set of a system exceeds the size of each level of the memory hierarchy, average data access latency increases roughly in steps [2].

The two most dramatic steps occur when the data set falls out of the cache and into main memory and when it falls out of main memory onto disk. Processes running on thin client system are by no means restricted to being interactive programs. In fact, multi-user systems like Unix are often used to run backgrounded compute or memory intensive jobs while the owner attends to other tasks.

As we saw in Section 4, poor resource scheduling of the processor can allow such greedy processes to severely impact perceived latency for interactive users. Likewise, poor resource management for memory can do the same. As observed by Evans et al., certain types of non-interactive, streaming memory jobs will typically force all other non-active processes to be paged to disk. They give as examples large data copies over NFS, creation of large temporary files in /tmp, and various stages of program compilation.

This behavior can be particularly damaging to interactivity in the following scenario. An interactive user may load a document into an application but then stop interacting with it for several minutes while he reads the document on-screen. During this time, a non-interactive

process on the same server with high page demand may force his application out to disk. When he goes to scroll down, there will be significant lag as his process is paged back into memory. We next demonstrate that TSE and Linux both perform poorly in this scenario.

In our tests, we opened a simple text editing application remotely. In Windows we used Notepad and in Linux *vim*. We then started on the server a process that touches enough memory to force pages of the edit application's memory to be swapped to disk. Then we input a single keystroke and measured the time it took for the server to respond with a screen update. As we saw in an earlier experiment, the response should come in less than 50ms. However, because the application's memory must be paged back from disk, significant latency is introduced. We report ranges and averages over ten runs for each operating system:

OS	Process Pages		
		In-Memory	Paged Out
Linux	<i>min</i>	50ms	330ms
	<i>avg</i>	50ms	1,170ms
	<i>max</i>	50ms	3,000ms
TSE	<i>min</i>	50ms	2,430ms
	<i>avg</i>	50ms	4,026ms
	<i>max</i>	50ms	11,850ms

These values are quite high and well into the range of perceptible latency. The latencies generated in TSE average about 40 times the threshold of human perception, while in Linux they average 11 times this limit.

Frequent paging of semi-active processes like we describe here indicates that the server has insufficient memory. A simple solution, of course, is to add more memory. Nevertheless, operating system mechanisms that mask scarcity-induced latency still provide at least two non-trivial benefits. First, a system that protects the pages of interactive processes is capable of hosting non-interactive batch jobs that have arbitrary memory requirements. A system without this protection cannot host such jobs without drastically degrading interactive performance. Second, paging protection guards against disruptive delays during spikes in memory utilization that are transient and do not warrant permanent additional resources.

Evans et al. also demonstrated in their prototype kernel a solution to this problem, which is non-interactive process throttling in high load situations. They demonstrated that their SVR4 kernel modified with throttling eliminated this pathology. The modifications they made could be adapted to today's TSE, Linux, and BSD kernels. But as with the scheduling modifications, the task would be easier in TSE than in Linux and BSD because the Unix/X platforms must rely on user-level applications to pro-

vide information about which threads and processes are latency-sensitive.

## 6 Network

In this section, we consider the impact of the network on latency and the role therein of operating system abstractions for display and input service.

First, we consider how user behavior generates network load. We compare the ability of RDP, X, and LBX to minimize network traffic for any given user behavior. The comparison includes a typical application workload, and an examination of the impact of trends in user interface design, particularly the growing usage of animation. Second, we discuss how network load translates to user-perceived latency, underscoring the importance of network protocol efficiency.

To simplify our discussion, we first define two terms. Let a "channel" simply be a directed stream of network messages between the client and server. We call the stream from the server to the client the display channel because it carries messages instructing the client to display application interface elements. The stream from the client to the server we call the input channel because it carries keystroke and mouse input information to the application.

### 6.1 From Behavior to Load

Thin client servers export user interfaces over network links to remote users. Therefore, load generated on the network resource depends heavily on the design and implementation of the user interfaces of the applications being run remotely.

Perhaps the most visible user application trend over recent years has been the increasing richness and sophistication of graphical interfaces. As a result, the typical user behavior in a thin client environment is becoming increasingly network intensive.

One of the strengths of TSE's design is that it allows existing Windows applications to run unmodified in a remote access environment. In fact, TSE would not be commercially viable if users could not use the same applications on which they currently rely. However, Windows software developers have typically designed their interfaces assuming fast, local graphics acceleration and therefore many applications that will be run on TSE will potentially consume unfriendly amounts of bandwidth.

Likewise, applications written for the X environment are growing more like modern Windows applications in their appearance and functionality, particularly with Linux's growing popularity on the desktop. In the following subsections, we investigate the network loads generated by typical modern graphical applications.

### 6.1.1 Compulsory Load

Compulsory network load includes both the quantity of bytes exchanged between the client and the server for session negotiation and initialization, and any network traffic that is exchanged after session setup but while the user is idle.

Session setup costs in our configurations were 45,328 bytes and 16,312 bytes for TSE and Linux/X, respectively. These costs are rare and ephemeral, and are typically not major contributors to latency. In terms of idle load, neither system requires data to be exchanged when no user activity is present. So compulsory load is a relative non-issue with the network on these two systems. The real contributor to latency is dynamic load generated by application usage, which we discuss next.

### 6.1.2 Dynamic Load

To gain a broad understanding of the relative performance of RDP, X, and LBX, we compared their behavior on a typical application workload. Thanks to the growth in popularity of Linux, we were able to develop a workload with cross-platform applications that run on Windows and Linux/X. These were Corel WordPerfect, a word processor, the Gimp, an open-source photo-editing package, and Netscape Navigator, a web browser.

For each network protocol, we performed a predefined set of user interactions: editing a WordPerfect document, creating a simple bitmap with the Gimp, and surfing several websites with Netscape. We collected data during these trials using *prototap*, our own protocol tracing software based on the *tcpdump pcap* packet sniffing library.

The following table shows byte and message counts for each channel and for each protocol. We also report the average message size for each protocol.

		RDP	LBX	X
Bytes	input	196,343	1,478,341	2,749,328
	display	2,049,132	8,137,445	14,183,842
	total	2,245,475	9,615,786	16,933,170
Msgs	input	1,155	18,888	19,345
	display	2,422	74,618	20,797
	total	3,577	93,506	40,142
Avg. msg size		627.75	102.84	421.83

RDP is clearly the most efficient protocol, generating less than 25% of the byte traffic of LBX and less than 15% of X. This is due in large part to the small number of messages it sends relative to X and LBX. However, RDP also has the largest average message size, suggesting that RDP messages encode a higher level of graphics semantics than do those of X and LBX. The message size

advantage of LBX over X is due to message compression. Note that this savings comes at the expense of a 133% increase in display message count over X. This, however, does not seem to adversely affect the overall performance of LBX.

The average message size among the three protocols is just 209 bytes, which is much smaller than the interface MTU on our systems (1500 bytes). For such small messages, the overhead imposed even by just 20 byte IP headers is significant. In non-routed deployment environments, a scheme like the *x-kernel* virtual-IP (VIP) network stack could reduce overhead by omitting the IP header [11]. The following table gives the potential byte savings of omitting the IP header.

	RDP	X	LBX
Normal Bytes	2,245,475	16,933,170	9,615,786
Bytes w/ VIP	2,169,435	16,096,790	7,745,666
Savings	3.38%	4.94%	19.45%

Because LBX has the smallest average message size, it stands to benefit most from a VIP-like scheme. However, even with this VIP optimization, LBX would still be less than half as efficient than RDP.

### 6.1.3 Animations

As discussed earlier, application interfaces have steadily grown more sophisticated and active. In particular, we observe an increasing use of animation in user interface design.

Animation is often employed to improve the user experience by creating the illusion of reduced latency through visual contiguity. Ironically, the use of smooth and effective animation in a thin client environment can produce considerable network load, yielding, on balance, a negative impact on user-perceived latency. Moreover, animations often run asynchronously of user interaction, meaning that their activity is not limited by the rate of user events such as keystrokes or mouse movements.

Simple animations like blinking cursors and progress bars generate a harmless amount of traffic, generally less than 10KBps for short durations. Other types of animation, however, can be quite costly. In particular, today's web pages are replete with animated GIF advertisements and Java and HTML based stock and news tickers. The remote display of the MSNBC homepage produces sustained average network loads well in excess of 1Mbps on RDP, X, and LBX.

Such levels of network activity make multi-user service over 10Mbps Ethernet unfeasible. If just five users open their browsers to a page like this, the network link becomes saturated. Although many administrators of interface service environments may, by policy, prohibit



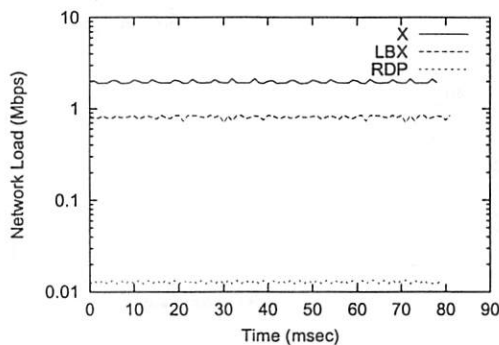


Figure 5: Network load generated by displaying a 100 by 100 pixel, 10-frame, 20Hz animated GIF in Netscape Navigator over X, LBX, and RDP.

the use of web browsers or enforce the disabling of webpage animations, this remains an important issue to consider when developing a “realistic” behavior profile for a user base. 100Mbps of faster Ethernet connectivity is virtually required to support this type of user behavior.

#### Taming Animation: Bitmap Caching

While on the animation-intensive MSNBC page all the protocols perform similarly, on a less intensive synthetic page, RDP significantly outperforms LBX and X.

Figure 5 shows the network load for displaying on Netscape a synthetic animated GIF. The animation is a 100 by 100 pixel rotating MSNBC logo that has 10 total frames and runs at 20 frames per second. Under LBX, bandwidth utilization is 0.8Mbps, while under X it is 2.0Mbps, and under RDP just 0.01Mbps.

This disparity between RDP and the X-based protocols suggests the presence of a client-side bitmap cache in TSE large enough to store all the frames of the synthetic animation. But for the animations on the more intensive MSNBC page, their competing frames overflow the cache such that each miss generates a full bitmap transfer over the network. When the frames do fit into the cache, we presume that display data not need to be transferred, so that only small “swap bitmap” messages are exchanged.

Indeed, according to Microsoft’s product literature, the TSE client reserves, by default, 1.5MB of memory for a bitmap cache using an LRU eviction policy [5]. The cache is typically used to store icons, button images, and glyphs. Storing these items can be especially bandwidth effective since users often spend much of their time in just a few applications each with a finite number of icons and images.

X, and consequently LBX, does not support bitmap caching, and the consequences are clearly shown in Fig-

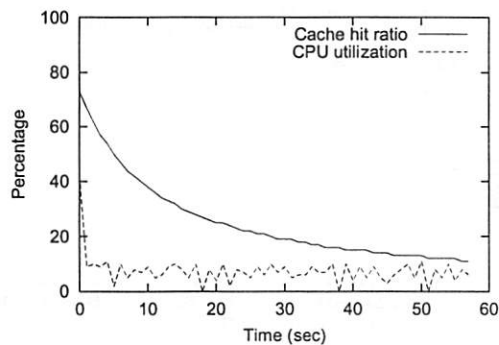


Figure 6: CPU utilization and bitmap cache hit ratio for a 66-frame animation that overflows the bitmap cache. The CPU hovers around 10% utilization. The hit ratio starts at 75% due to cache hits on screen elements displayed before the beginning of the test. During the test, the oversize animation overflows the cache, preventing any hits, and resulting in an asymptotically declining cumulative cache hit ratio.

ure 5. If there were a cache of any appreciable size (which there is not) it is not being used. Each frame of the animation requires the full bitmap to be transferred across the network.

The difference in performance between RDP and X/LBX again reinforces the importance of considering the design and implementation of the operating system abstractions for hardware resources. The inclusion of a bitmap cache in TSE’s abstraction for display and input allows it to handle behavior that includes animation with much less load. That, in turn, means that TSE can support more concurrent users with this behavior before exhibiting noticeable latency.

#### Cache Effectiveness and CPU Load

The effectiveness of the cache is not only critical to reducing network load, but also processor load at the server. In these tests, we used the Session object in the Microsoft Performance Monitor to measure the various metrics associated with the client-side bitmap cache.

Figure 6 shows the effect on CPU Utilization and Bitmap Cache Hit Ratio of a 66-frame animation that overflows the cache. The CPU Utilization starts at around 10% as it transmits frames for the first time. However, it never falls, because the server must continue to send the frames that fall out of the cache just before being needed, which is all of them. The Cache Hit Ratio which is cumulative, begins around 70%, and falls asymptotically toward zero with each subsequent miss.

#### Cache Pathology

However, bitmap caching has its limitations. Loop-

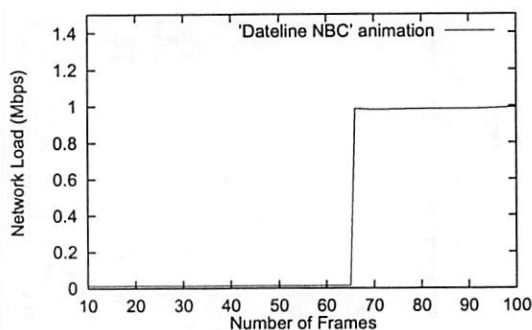


Figure 7: Network load generated as IE4 displays animations of varying frame counts over RDP. For frame counts less than 65, the animation fits in the cache, and network load is negligible. For 66 or more frames, the animation overflows the cache and every frame must be transferred over the wire, generating considerable load.

ing animations defeat LRU bitmap caches in the same way that sequential byte range accesses defeat LRU disk caches, which is a well-known phenomenon in file systems research [15].

To demonstrate, we created a series of animations whose frame counts range from 25 to 100. Figure 7 shows that for values 25 through 65, bandwidth utilization is 0.01Mbps, but for all values above 65, bandwidth utilization is 0.96Mbps.

Clearly, while LRU may be the appropriate eviction scheme for typical usage, it is exactly the wrong scheme for handling looping animations. A more intelligent scheme capable of dealing with such animations might somehow detect loop patterns and adjust its eviction behavior accordingly.

X Windows could certainly benefit from the introduction of client-side bitmap caching, and even TSE could benefit from a larger cache. Even though Microsoft claims that “testing has found that 1.5 MB is the optimal cache size,” our results show that modest amounts of interface animation can render a cache of this size useless [5]. When available, additional memory should be allocated to enlarge the bitmap cache. As a rule of thumb, a 256-color, 468 by 60 pixel banner advertisement with 10 seconds of 20 frame per second animation requires about 5.6 MB of memory to cache the entire animation. This figure does not account for any compression that is possible on the frames, and so serves as an upper bound for this amount of animation.

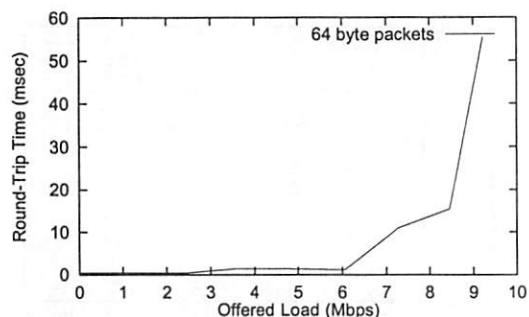


Figure 8: Latency increases above human-perceptible levels as the network fabric approaches saturation.

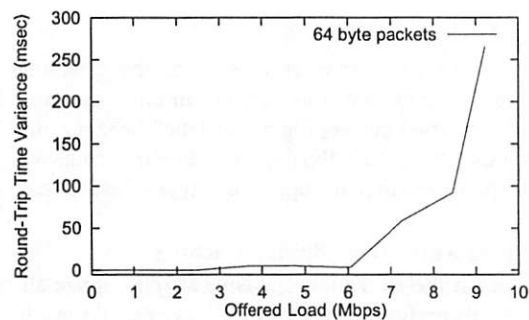


Figure 9: Jitter also increases considerably as the network fabric approaches saturation.

## 6.2 From Load to Latency

We have just shown how various user behaviors generate network load. The next step in the framework is mapping network load levels to user-perceived latency.

To investigate this relationship, we produced synthetic TCP/IP network load on our experimental testbed. The load was generated by two simple C programs that establish a TCP connection and send and receive random data at various rates. Figures 8 and 9 show the effect of load on network latency and jitter. For each load level, we ran *ping* for 60 seconds and took the average and variance in round-trip time (RTT) for all packets sent. We used the default packet size in *ping*, which is 64 bytes. 64 bytes is roughly the size of a typical input channel message, such as a keystroke. Therefore, the latencies we observe here give a realistic lower-bound for latencies that would be observed by a user.

Figure 9 shows that while the network is not saturated, RTT remains low and almost perfectly consistent. However, as the network nears saturation, performance suffer-

s dramatically. The 55ms delay induced at 9.6Mbps load is considerable with respect to known levels of human latency tolerance [7]. The inconsistency of the latency, a phenomenon known as jitter, only compounds the negative impact of network saturation.

## 7 Related Work

There is relatively little other work directly relevant to our broad discussion of operating system design impact on thin client performance. What there is, we now discuss and analyze in light of our approach and findings.

Endo et al. and Evans et al. have emphasized the importance of latency, and were discussed in detail in Sections 4 and 5.

Danskin published several papers on profiling the X protocol [6]. His work, in terms of our framework, focused primarily on determining the load generated by typical applications of his day. While the typical X behavior profile has changed considerably since then, his methodology provides the inspiration for our *prototap* tool. Danskin also did work on characterizing application-specific idioms used on the display channel. Finally, he came to the same conclusion as we did that small message size makes TCP/IP an inefficient network substrate for protocols like RDP, X, and LBX.

Schmidt et al. introduced a new thin client wire protocol called SLIM, which is embodied in the Sun Microsystems SunRay product [19]. While SLIM has the advantage of being more platform independent than X or RDP, their results show it to be roughly equivalent in performance to X, placing it still behind RDP and LBX in network load efficiency. Their paper also touches upon the issue of latency induced by server-side resource contention, but omits analysis of how the server-side operating system can be tuned to reduce latency. We contend that in the long view, network protocol efficiency is just one piece of a high-performance, low-latency thin client user experience. VNC, from AT&T Laboratories Cambridge, is another network protocol that is similar to SLIM [16].

Sirer et al. describe an architecture for extruding the components of a virtual machine over multiple hosts in a network [22]. Their work further emphasizes the benefits of factoring system services out of clients and onto network servers. Their DVM architecture differs from what are colloquially called thin client architectures in that it draws the line where the network is spliced in a different place, factoring out services such as verification, security enforcement, compilation and optimization, but not execution. Moreover, their work emphasizes throughput and not user interaction latency, and focuses more on the network bandwidth impact of code transfers rather than user interface input and display traffic.

On the performance of TSE, the only documents we have been able to find are server sizing white papers published by Microsoft and various hardware vendors who market TSE servers [14, 10, 3]. These white papers are remarkably similar, defining typical user profiles and reporting the load generated by these profiles. They uniformly ignore, however, the issue of user-perceived latency. We also believe the network load characterizations in these papers are overly optimistic, ignoring the increasingly dynamic and rich nature of user interfaces.

## 8 Conclusion

Latency is the paramount performance criterion for operating system support for thin client service, or interactive, graphical, multi-user, remote access.

We have presented an approach for evaluating thin client environments which is founded on latency and highlights important issues relevant to thin client performance. These include the influence of user-specific behavior, the translation of that behavior into resource load, the importance of operating system abstraction implementation therein, and the translation of resource load into user-perceived latency.

This approach guided our resource-by-resource comparison of TSE and X Windows on Linux, two popular implementations of thin client services. Our investigation reveals that resource scheduling for both the processor and memory in these systems is not well optimized for heavy, concurrent, interactive use. In common cases of resource saturation, both latency and jitter rise well above human-perceptible levels. We also performed a detailed comparison of the RDP, X, and LBX protocols and found that RDP is generally more efficient in terms of network load, particularly in handling animated user interface elements.

## 9 Acknowledgements

We would like to thank our shepherd, Christopher Small, and the anonymous members of the program committee for their help in shaping this paper. We also thank Mike Smith and H.T. Kung for their review and commentary on early incarnations of this work.

## References

- [1] Beck, M., Bohme, H., Dziadzka, M., Kunitz, U., Magnus, R., Verworner, D. *Linux Kernel Internals: Second Edition*, Addison-Wesley Longman, 1998.
- [2] Chen, J.B., Endo, Y., Chan, K., Mazieres, D., Dias, A., Seltzer, M., Smith, M. "The Measured Performance of Personal Computer Operating Systems."

- ACM Transactions on Computer Systems, Vol. 14, No. 1, February 1996, Pages 3-40.
- [3] Compaq Corp. "Performance and Sizing of Compaq Servers with Microsoft Windows NT Server 4.0, Terminal Server Edition". White Paper. July 1998.
  - [4] Conner, B., Holden, L. "Providing A Low Latency User Experience In A High Latency Application". Proceedings of the 1997 Symposium on Interactive 3D Graphics.
  - [5] Cumberland, C., Carius, G., Muir, A. *Microsoft Windows NT Server 4.0 Terminal Server Edition Technical Reference*, Microsoft Press, 1999.
  - [6] Danskin, J., Hanrahan, P. "Profiling the X Protocol". In Proceedings of the 1994 SIGMETRICS Conference on Measurement and Modeling of Computer Systems.
  - [7] Endo, Y., Wang, Z., Chen, B., Seltzer, M. "Using Latency to Evaluate Interactive System Performance". Proceedings of the 1996 Symposium on Operating System Design and Implementation(OSDI).
  - [8] Evans, S., Clarke, K., Singleton, D., Smaalders, B., "Optimizing Unix Resource Scheduling for User Interaction," 1993 Summer USENIX Conference.
  - [9] Fulton, J., Kantarjiev, C. "An Update on Low Bandwidth X(LBX): A Standard for X and Serial Lines." Proceedings of the 7th Annual X Technical Conference.
  - [10] Hewlett-Packard Corp. "Performance and Sizing Analysis of the Microsoft Windows Terminal Server on HP NetServers". White paper. July 1998.
  - [11] Hutchinson, N., Peterson, L., Abbott, M., O'Malley, S. "RPC in the x-Kernel: Evaluating New Design Techniques". SOSP 1989: 91-101.
  - [12] Li, K., and Hudak, P., "Memory Coherence in Shared Virtual Memory Systems." ACM Transactions on Computer Systems, November 1989.
  - [13] MacKenzie, I., and Ware, C. "Lag as a Determinant of Human Performance in Interactive System." Proceedings of the ACM InterCHI '93.
  - [14] Microsoft Corp. "Microsoft Windows NT Server 4.0, Terminal Server Edition — Capacity Planning". White Paper. June 1998.
  - [15] Patterson, R., Gibson, G., Ginting, E., Stodolsky, D., Zelenka, J., "Informed Prefetching and Caching," Proceedings of the Fifteenth Symposium on Operating Systems Principles, December 1995, pp. 79-95.
  - [16] Richardson, T., Stafford-Fraser, Q., Wood, K., Hopper, A., "Virtual Network Computing", IEEE Internet Computing, Jan/Feb 1998.
  - [17] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyon, B., "Design and Implementation of the Sun Network Filesystem," Proceedings of the Summer 1985 USENIX Conference.
  - [18] Scheifler, R., Gettys, J. "X Window System: Core and Extension Protocols". Butterworth-Heinemann. 1997
  - [19] Schmidt, B., Lam, M., Northcutt, J., "The interactive performance of SLIM: a stateless, thin client architecture," Operating Systems Review, 34(5):32-47, December 1999.
  - [20] Shneiderman, B. "Designing the User Interface". Addison-Wesley. 1992.
  - [21] Solomon, D. *Inside Windows NT: Second Edition*, Microsoft Press, 1998.
  - [22] Sirer, E.M., Grimm, R., Gregory, A., Bershad, B., "Design and implementation of a distributed virtual machine for networked computers," SOSP 1999: 202-216.
  - [23] Wang, Z., Rubin, N. (1998) "Evaluating the Importance of User-Specific Profiling," Proceedings of the 3rd USENIX Windows NT Symposium.



# Techniques for the Design of Java Operating Systems

Godmar Back Patrick Tullmann Leigh Stoller Wilson C. Hsieh Jay Lepreau

*Department of Computer Science  
University of Utah*

## Abstract

Language-based extensible systems, such as Java Virtual Machines and SPIN, use type safety to provide memory safety in a single address space. By using software to provide safety, they can support more efficient IPC. Memory safety alone, however, is not sufficient to protect different applications from each other. Such systems need to support a *process model* that enables the control and management of computational resources. In particular, language-based extensible systems should support resource control mechanisms analogous to those in standard operating systems. They need to support the separation of processes and limit their use of resources, but still support safe and efficient IPC.

We demonstrate how this challenge is being addressed in several Java-based systems. First, we lay out the design choices when implementing a process model in Java. Second, we compare the solutions that have been explored in several projects: Alta, K0, and the J-Kernel. Alta closely models the Fluke operating system; K0 is similar to a traditional monolithic kernel; and the J-Kernel resembles a microkernel-based system. We compare how these systems support resource control, and explore the tradeoffs between the various designs.

## 1 Introduction

Language-based extensible systems in the form of Java virtual machines are used to implement execution environments for applets in browsers, servlets in servers, and mobile agents. All of these environments share the property that they run multiple applications at the same time. For example, a user may load applets from different Web sites into a browser; a server may run servlets from different sources; and an agent server may run agents from across the Internet. In many circumstances

This research was supported in part by the Defense Advanced Research Projects Agency, monitored by the Department of the Army under contract number DABT63-94-C-0058, and the Air Force Research Laboratory, Rome Research Site, USAF, under agreement numbers F30602-96-2-0269 and F30602-99-1-0503.

Contact information: {gback,tullmann,stoller,wilson,lepreau}@cs.utah.edu. Department of Computer Science, 50 S. Central Campus Drive, Room 3190, University of Utah, SLC, UT 84112-9205. <http://www.cs.utah.edu/flux/java/>

these applications can not be trusted, either by the server or user that runs them, or by each other. Given untrusted applications, a language-based extensible system should be able to isolate applications from one another because they may be buggy or even malicious. An execution environment for Java byte code that attempts to provide such isolation is what we term a "Java operating system."

Conventional operating systems provide the abstraction of a *process*, which encapsulates the execution of an application. A *process model* defines what a process is and what it may do. The following features are necessary in any process model for safe, extensible systems:

- **Protection.** A process must not be able to destroy the data of another process, or manipulate the data of another process in an uncontrolled manner.
- **Resource Management.** Resources allocated to a process must be separable from those allocated to other processes. An unprivileged or untrusted process must not be able to starve other processes by denying them resources.
- **Communication.** Since an application may consist of multiple cooperating processes, processes should be able to communicate with each other. Supported communication channels must be safe and should be efficient.

These requirements on processes form one of the primary tradeoffs in building operating systems, as illustrated in Figure 1. On the right-hand side, processes can be protected from each other most easily if they are on completely separate machines. In addition, managing computational resources is much simpler, since the resources are completely separate. Unfortunately, communication is more expensive between processes on different machines. On the left-hand side, communication is much cheaper, since processes can share memory directly. As a result, though, protection and accurate resource accounting become more difficult.

Operating systems research has spanned the entire range of these systems, with a primary focus on systems in the middle. Research in distributed systems and networking has focused on the right side of the figure. Research on single-address-space operating systems such as

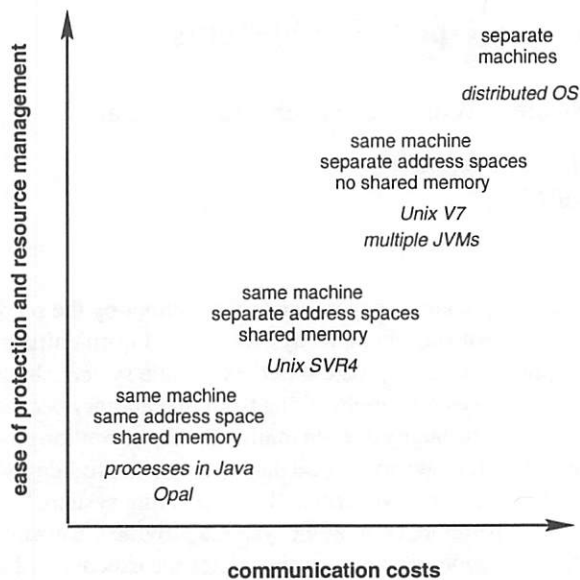


Figure 1: Trading off sharing and isolation between processes. On the right, running different processes on separate machines isolates them cleanly, but communication is more expensive. On the left, in theory a single-address-space operating system allows the most efficient communication between processes, but isolation is the most difficult.

Opal [13], as well as older work on language-based operating systems [38, 43] has focused on the left side of the figure. The reemergence of language-based extensible systems, such as SPIN [9, 32, 51] has focused attention back on the left side of the diagram. Such systems are single-address-space systems that use type safety instead of hardware memory mapping for protection. In this paper we discuss how resource management can be provided in language-based systems (in particular, in Java), and how the tradeoff between memory control and sharing is expressed in these systems.

We view Java as an example language-based extensible system for several reasons. First, Java's use of load-time bytecode verification removes the need for a trusted compiler. Second, Java's popularity makes it possible that a process model could be used widely. Finally, Java is general enough that the lessons we have learned in developing a process model for it should apply to other language-based extensible systems.

Systems such as Servlet Engines [4] and mobile agent servers [30] need to support multiple Java applications simultaneously. For safety, these systems use separate Java virtual machines to contain each application. While it is possible to run multiple Java applications and applets in separate Java virtual machines (JVMs), there are several reasons to run them within a single virtual ma-

chine. Aside from the overhead involved in starting multiple JVMs, the cost of communication between applications and applets is greater when applications are run in separate virtual machines (as suggested by Figure 1). Additionally, in small systems, such as the PalmPilot, an OS or hardware support for multiple processes might not be present. In such environments, a JVM must perform operating system tasks. A final reason to use a single JVM is that better performance should be achievable through reduction of context switching and IPC costs. Unfortunately, standard Java systems do not readily support multiprogramming, since they do not support a process abstraction. The research issues that we explore in this paper are the design problems that arise in implementing a process model in a Java virtual machine.

The hard problems in implementing a process model for Java revolve around memory management. Other hard problems in designing a process model are not unique to Java. In a Java system, protection is provided through the type safety of the language. Memory management is harder in Java than in conventional operating systems because the address space is shared. In a conventional operating system, protection is provided through a memory management unit. Process memory is inherently separated, and systems must be engineered to provide fast, efficient communication.

In this paper we compare several Java systems, and the process models that they support: Alta and K0, two projects at the University of Utah, and the J-Kernel, a project at Cornell. Alta is structured much like the Fluke microkernel [22], provides a hierarchical process model, and focuses on providing safe, efficient sharing between processes with potentially different type-spaces. K0 is structured much like a traditional monolithic kernel and focuses on stringent and comprehensive resource controls. The J-Kernel is structured like a microkernel-based system, with automatic stub generation for inter-task communication. It should not be surprising that language-based operating systems can adopt ideas from previous operating systems research: many of the design issues and implementation tactics remain the same. These systems support strong process models: each can limit the resource consumption of processes, but still permit processes to share data directly when necessary.

Section 2 overviews Java and its terminology. Section 3 describes the technical challenges in providing resource management in Java, and Section 4 compares the design and implementation of Alta, K0, and the J-Kernel. Section 5 describes related research in traditional operating systems, language-based operating systems, and Java in particular. Section 6 summarizes our conclusions.

## 2 Background

Java is both a high-level object-oriented language [26] and a specification for a virtual machine that executes bytecodes [32]. Java gives applications control over the dynamic linking process through special objects called *class loaders*. Class loaders support user-defined, type-safe [31] loading of new data types, object types, and code into a running Java system.

A JVM is an architecture-neutral platform for object-oriented, multi-threaded applications. The JVM provides a number of guarantees, backed by run-time verification and automatic memory management, about the memory safety of applications it executes. Specifically, the bytecodes that constitute an application must satisfy certain semantic constraints, and only the JVM-provided automatic garbage collector can reclaim storage.

A traditional JVM is structured as a trusted core, usually implemented in C, augmented with Java libraries. Together, the core and libraries implement the standard Java class libraries. Calls to the core C code are made through *native methods*.

Protecting a system from buggy or malicious code, and protecting clients from each other, requires more control than just the protection afforded by type safety. In particular, a JVM must also be able to provide security (control over data, such as information in files) and resource management (control over computational resources, such as CPU time and memory). A JVM is therefore analogous to a traditional operating system.

Although extensive investigation has been devoted to security issues in Java [25, 47], resource management has not been as thoroughly investigated. For example, a client can abuse its use of memory (either intentionally or accidentally) to compromise the overall functionality of a JVM. The design and implementation of robust Java operating systems that tightly control resource usage is an open area of research that we are addressing.

## 3 Resource Management

This section discusses the primary design choices for managing resources in a Java operating system. We divide the problem of resource management into three related subproblems:

- *Resource accounting*: the ability to track resource usage. Accounting can be exact or approximate, and can be fine-grained or coarse-grained.
- *Resource reclamation*: the ability to reclaim a process's resources when it terminates. Complex allocation management policies and flexible sharing policies can make reclamation difficult. Reclamation can be immediate or delayed.

- *Resource allocation*: the ability to allocate resources to processes in a way that does not allow processes to violate imposed resource limits. Allocation mechanisms should be fair and should not incur excessive overhead.

We discuss each of the previous issues with respect to several computational resources: memory, CPU usage, and network bandwidth. We do not currently deal with managing the use of persistent storage, because there is little specific to the management of such storage in language-based systems. Since Java encourages direct sharing of memory, the primary difficulty in supporting a process model in Java is in isolating processes' memory from one another.

### 3.1 Memory

The issues of memory accounting, memory reclamation and memory allocation within a Java process model can be divided into two discussions: memory accounting and the impact of the inter-process sharing model; and allocation and deallocation policies.

#### 3.1.1 Sharing Model

A *sharing model* defines how processes can share data with each other. In a Java operating system, three choices are possible: copying, direct sharing, and indirect sharing. The sharing model in standard Java (without processes) is one of *direct sharing*: objects contain pointers to one another, and a thread accesses an object's fields via offsets from the object pointer. In Java with processes, the choice of sharing model affects how memory accounting and process termination (resource reclamation) can be implemented.

**Copying.** Copying is the only feasible alternative when address spaces are not shared: for example, when two processes are on different machines. Copying was the traditional approach to communication in RPC systems [10], although research has been aimed at reducing the cost of copying for same-machine RPC [8]. Mach [1], for example, used copy-on-write and out-of-line data to avoid extra copies.

If data copying is the only means of communication between processes, then memory accounting and process termination are straightforward. Processes do not share any objects, so a process's objects can be reclaimed immediately; there can be no ambiguity as to which process owns an object. Of course, the immediacy of reclamation depends on the garbage collector's involvement in memory accounting: reclaiming objects in Java could require a full garbage collection cycle.

In Java, the use of copying as a communication mechanism is unappealing because it violates the spirit of the Java sharing model, and because it is slow. There



is enough support in a JVM for one process to safely share a trusted object with an untrusted peer; not leveraging this support for fine-grained sharing in a Java process model neutralizes the major advantage of using a language-based system. On the other hand, in a system that only supports copying data between processes, process termination and per-process memory accounting are much simpler.

**Direct Sharing.** Since Java is designed to support direct sharing of objects, another design option is to allow direct sharing *between* processes. Interprocess sharing of objects is then the same as intraprocess sharing. Direct sharing in single-address-space systems is somewhat analogous to shared memory (or shared libraries) in separate-address-space systems, but the unit of sharing is much finer-grained.

If a system supports direct sharing between processes, then process termination and resource reclamation are complicated. If a process exports a directly shared object, that object cannot be reclaimed when the exporting process is terminated. The type-safety guarantees made by the Java virtual machine cannot be violated, so any reference to the object must remain valid. To reclaim the object would require that all references to the object be located. In the presence of C code, such a search is impossible to do without extensive compiler support. Therefore, in order to support resource reclamation when a process is killed, either direct sharing needs to be restricted or the system must guarantee that all outstanding references to any object can be located.

**Indirect Sharing.** An alternative to direct sharing is *indirect sharing*, in which objects are shared through a level of indirection. When communicating a shared object, a direct pointer to that object is not provided. Instead, the process creates a proxy object, which encapsulates a reference to the shared object. It then passes a pointer to the proxy object. Proxies are system-protected objects. In order to maintain indirect sharing (and prevent direct sharing), the system must ensure that there is no way for a client to extract a direct object pointer from a proxy. Such second-class handles on objects are commonly called capabilities; analogues in traditional operating systems include file descriptors and process identifiers.

Compared to direct sharing, indirect sharing is less efficient, since an extra level of indirection must be followed whenever an interprocess call occurs. The advantage of indirection, however, is that resource reclamation is straightforward. All references to a shared object can be revoked, because the level of indirection enables the system to track object references. Therefore, when a process is killed, all of its shared objects can be reclaimed immediately. As with copying, immediate revocation is subject to the cost of a full garbage collection cycle in

Java.

### 3.1.2 Allocation and Deallocation

Without page-protection hardware, software-based mechanisms are necessary to account for memory in a Java operating system. Every allocation (or group of allocations) must be checked against the allocating process's heap limit. Stack frame allocations must be checked against the executing thread's stack limits.

Memory is necessarily reclaimed in Java by an automatic garbage collector [50]. It seems obvious to use the garbage collector to do memory accounting as well. The simplest mechanism for keeping track of memory is to have the allocator debit a process that allocates memory, and have the garbage collector credit a process when its memory is reclaimed.

In the presence of object sharing (whether direct or indirect), other memory accounting schemes are possible. For example, a system could conceivably divide the "cost" of an object among all the parties that keep the object alive. This model has the drawback that a process can be spontaneously charged for memory. For example, suppose a process acquires a pointer to a large object, and is initially only charged for a small fraction of the object's memory. If the other sharers release their references, the process may asynchronously run out of memory, because it will be forced to bear the full cost of the entire object.

Another potential scheme is to allow processes to pass memory "credits" to other processes. For example, a server could require that clients pass several memory credits with each request to pay for the resources the server allocates. Such a scheme is analogous to economic models that have been proposed for resource allocation [46]. A similar system might permit a process to transfer the right to allocate under its allowance. A similar effect is possible in the simple allocator-pays model by having a client allocate an object and pass it to the server to be "filled in."

An important issue in managing memory is the relationship between allocation and accounting schemes. In particular, a system that charges per object, but allocates memory in larger chunks, might be subject to a fragmentation attack. A process with a small budget could accidentally or maliciously cause the allocation of a large number of blocks. One solution is to provide each process with its own region of physical or virtual addresses from which to allocate memory. While this solution guarantees accurate accounting for internal fragmentation, it may introduce external fragmentation.

## 3.2 CPU Usage

The two mechanisms necessary for controlling CPU usage are accounting and preemption. The system must



be able to account accurately for the CPU time consumed by a thread. The system must also be able to prevent threads from exceeding their assigned CPU limits by preempting (or terminating) them. Desirable additional features of CPU management are multiple scheduling policies, user-providable policies, and support for real-time policies.

### 3.2.1 CPU Accounting

The accuracy of CPU accounting is strongly influenced by the way in which processes obtain services. If services are implemented in libraries or as calls to a monolithic kernel, accounting simply amounts to counting the CPU time that a thread accrues.

CPU accounting is difficult with shared system services, where the process to bill for CPU usage is not easily determined. Examples of such services include garbage collection and interrupt processing for network packets. For such services, the system needs to have a means of deciding what process should be charged.

**Garbage Collection.** The simplest accounting policy for garbage collection is to treat it as a global system service. Unfortunately, such a policy is undesirable because it opens the system to denial-of-service attacks. For example, a process could trigger garbage collections frequently so as to slow down other processes. In addition, treating garbage collection as a universal service allows priority inversion to occur. If a low-priority thread allocates and deallocates large chunks of memory, it may cause a high-priority thread to wait for a garbage collection.

We see two approaches that can be taken to solve this problem. First, the garbage collector could charge its CPU usage to the process whose objects it is traversing. However, since this solution would require fine-grained measurement of CPU usage, its overhead would likely be prohibitive.

The second alternative is to provide each process with a heap that can be garbage-collected separately, such that the GC time can be charged to the owning process. Independent collection of different heaps requires special treatment of inter-heap references if direct sharing is to be allowed. In addition, distributed garbage collection algorithms might be necessary to collect data structures that are shared across heaps.

**Packet Handling.** Interrupt handling is another system service, but its behavior differs from that of garbage collection, because the “user” of an external interrupt cannot be known until the interrupt is serviced. The goal of the system should be to minimize the time that is needed to identify the receiver, as that time cannot be accounted for. For example, Druschel and Banga [20] showed how packets should be processed by an operating system. They demonstrated that system performance

can drop dramatically if too much packet processing is done at interrupt level, where normal process resource limits do not apply. They concluded that systems should perform *lazy receiver processing* (LRP), which is a combination of early packet demultiplexing, early packet discard, and processing of packets at the receiver’s priority. The use of LRP improves traffic separation and stability under overload.

### 3.2.2 Preemption and Termination

Preempting a thread that holds a system lock could lead to priority inversion. As a result, it is generally better to let the thread exit the critical section before it is preempted. Similarly, destroying a thread that holds a system lock could lead to consistency problems if the lock is released or deadlock if the lock is never released. Preemption and termination can only be safe if the system can protect critical sections against these operations.

By making a distinction between non-preemptible, non-killable code, and “regular” code a Java system effectively makes a distinction between user-mode and kernel-mode [5]. In a traditional, hardware-based system, entry to (and exit from) the kernel is explicit: it is marked with a trap instruction. The separation between kernel and user code is not as clear in Java, since making a call into the kernel might be no different than any other method invocation.

In addition to providing support for non-preemptible (and non-killable) critical sections, a Java operating system needs to have a preemption model within its kernel. The design choices are similar to those in traditional systems. First, the kernel could be single-threaded, and preemption would only occur outside the kernel. Alternatively, the system can be designed to allow multiple user threads to enter the kernel. In the latter case, preemption might be more immediate, but protecting the kernel’s data structures incurs additional overhead.

## 3.3 Network Bandwidth

While bandwidth can be important for certain applications of Java, such as active networks [48], there is little about controlling network bandwidth that is specific to Java. A range of approaches, from byte counting to packet scheduling, is available.

The J-Kernel experimented with a special version of the WinSocket DLL to count bytes in outgoing network streams. The implementations of K0 and Alta could easily provide access to packet scheduling facilities provided by the infrastructure on which they run.

## 4 Comparison

In this section we describe in detail our two prototype systems, K0 and Alta, and the prototype of a third

Java operating system, J-Kernel, that has been built at Cornell. These systems represent different sets of design tradeoffs:

- The J-Kernel disallows direct sharing between processes, and uses bytecode rewriting to support indirect sharing. Because it consists of Java code only, it is portable across JVMs. As a result, though, the resource controls that the J-Kernel provides are approximate. J-Kernel IPC does not involve a rendezvous: a thread migrates across processes, which can delay termination.
- KO partitions the Java heap so as to isolate resource consumption. In addition, restricted direct sharing is permitted through the system heap. Garbage collection techniques are put to interesting use to support this combination. CPU inheritance scheduling is used as a framework for hierarchical scheduling of CPU time.
- Alta uses hierarchical resource management, which makes processes responsible for (and gives them the capability of) managing their subprocesses' resources. Direct sharing between sibling processes is permitted because their parent is responsible for their use of memory. The hierarchy also is a good match for CPU inheritance scheduling.

## 4.1 J-Kernel

The J-Kernel [15, 28] implements a microkernel architecture for Java programs, and is itself written in Java. It supports multiple protection domains that are called tasks. Names are managed in the J-Kernel through the use of *resolvers*, which map names onto Java classes. When a task creates a subtask, it can specify which classes the subtask is allowed to access. Class loaders are used to give tasks their own name spaces.

### 4.1.1 System Model

Communication in the J-Kernel is based on capabilities. Java objects can be shared indirectly by passing a pointer to a *capability* object through a "local RMI" call. The capability is a trusted object containing a direct pointer to the shared object. Because of the level of indirection through capabilities to the shared object, the capabilities can be revoked. A capability can only be passed if two tasks share the same class. Making a class shared is an explicit action that forces two class loaders to share the class.

All arguments to inter-task invocations must either be capabilities, or be copied in depth, i.e., the complete tree of objects that are reachable from the argument via

direct references must be copied recursively. By default, standard Java object serialization is used, which involves marshaling into and unmarshaling from a linear byte buffer. To decrease the cost of copying, a fast copy mechanism is also provided. Specialized code for a class creates a direct copy of an object's fields. Both the specialized fast copy code and the stubs needed for cross-domain calls are generated dynamically.

The J-Kernel supports thread migration between tasks: cross-task communication is not between two threads. Instead, a single thread makes a method call that logically changes protection domains. Therefore, a full context switch is not required. To prevent malicious callers from damaging a callee's data structures, each task is only allowed to stop a thread when the thread is executing code in its own process. This choice of system structure requires that a caller trust all of its callees, because a malicious or erroneous callee might never return.

### 4.1.2 Resource Management

The J-Kernel designers made the explicit decision not to build their own JVM. Instead, the J-Kernel is written entirely in Java. As a result of this decision, the J-Kernel designers limited the precision of their resource control mechanisms. The lack of precision occurs because the JVM that runs under the J-Kernel cannot know about processes. As a result, it cannot account for the resources that it consumes on behalf of a process.

**Memory Management.** In order to account for memory, the J-Kernel rewrites the bytecode of constructors and finalizers to charge and credit for memory usage. Such a scheme does not take fragmentation into account. In addition, memory such as that occupied by just-in-time compiled code is hard to account for.

**CPU Management.** The NT version of the J-Kernel uses a kernel device driver to monitor the CPU time consumed by a thread. This mechanism is reactive: threads can only be prevented from consuming further resources after they already exceeded their limits. In addition, it is difficult to add custom scheduling policies for tasks.

### 4.1.3 Implementation Status

A version of the J-Kernel that does not support resource controls is freely available from Cornell [45]. The advantage of their implementation approach is a high degree of portability: the J-Kernel can run on most JVMs. Since it uses class reloading, there are some dependencies on the specific interpretation of gray areas in the Java language specification.

The J-Kernel is distributed with two additional pieces of software. The first is JOS, which uses the J-Kernel to provide support for servers. The second is J-Server, a Web server that safely runs client-provided Java code.

#### 4.1.4 Summary

The J-Kernel adopts a capability-based model that disallows direct sharing between tasks. As a result, its capabilities are directly revocable, and memory can be completely reclaimed upon task termination. In addition, the J-Kernel exploits the high-level nature of Java's byte-code representation to support the automatic creation of communication channels.

## 4.2 K0

K0's design loosely follows that of a traditional monolithic kernel. K0 is oriented toward complete resource isolation between processes, with the secondary goal of allowing direct sharing. As in a traditional operating system, each process is associated with a separate heap, and sharing occurs only through a special, shared heap.

K0 can run most JDK 1.1 applications without modification. It cannot run those that assume that they were loaded by a "null" class loader.

### 4.2.1 System Model

A K0 process consists of a name space, a heap, and a set of threads. K0 relies on class loaders to provide different processes with separate name spaces. Each process is associated with its own class loader, which is logically considered part of the kernel. To provide different processes with their own copies of classes that contain static members, K0 loads classes multiple times. Unlike other JVMs, K0 allows safe reloading of all but the most essential classes, such as `Object` or `Throwable`. To reduce a process's memory footprint, classes that do not contain shared data may be shared between processes, akin to how different processes map the same shared library into their address spaces in a traditional OS. However, since all shared classes must occupy a single name space, sharing is a privileged operation.

Threads access kernel services by calling into kernel code directly. The kernel returns references to kernel objects that act as capabilities to such things as open files and sockets. In order to support the stopping or killing of threads, K0 provides a primitive that defers the delivery of asynchronous exceptions until a well-defined cancellation point within the kernel is reached. This primitive does not solve all of the problems with thread termination, but it enables the kernel programmer to safely cancel user processes without compromising the integrity of the kernel.

Each K0 process is associated with its own heap. Shared classes and other shared data reside in a distinct heap called the shared heap. K0 supports comprehensive memory accounting that takes internal allocations by the JVM into account. Because K0 controls inter-heap references, it is able to support independent collec-

tion of individual heaps and it is able to charge garbage collection time to the appropriate processes. The use of separate heaps has the additional benefit of allowing K0 to avoid priority inversions: it is not necessary to stop higher-priority threads in other processes when performing a collection.

### 4.2.2 Resource Management

**Memory Management.** The use of separate heaps simplifies memory accounting because each heap is subject to its own memory budget, and simplifies CPU accounting because each heap can be collected separately. In order to preserve these benefits while still allowing for efficient process communication, K0 provides limited direct sharing between heaps. If two processes want to share an object, two criteria must be met. First, the processes must share the type of the object. Second, the object must be allocated in the shared heap. The creation of a shared object is a privileged operation. An object in a process heap can refer to a shared object, and a shared object can refer to an object in a process heap. However, K0 explicitly disallows direct sharing between objects in separate processes' heaps, and uses write barriers [50] to enforce this restriction.

Acquiring a reference to a shared object is only possible by invoking the system, and K0 ensures that resources allocated within the system heap on behalf of an process are subject to a specific limit. For instance, each process may only open a certain number of files, since the kernel part of a file descriptor is allocated in system space. K0 must be careful to not hand out references to objects that have public members, or objects it uses for internal synchronization.

Shared objects have a restricted execution model. During their construction, they can allocate objects on the system heap. After the objects are constructed, threads that methods on them are subject to normal segment limits: if a thread attempts to use a shared object to write a reference to a foreign heap into its own heap, a segmentation violation error will be triggered.

To allow for separate garbage collection of individual heaps, K0 implements a form of distributed GC [37]. For each heap, K0 keeps a list of *entry items* for objects to which external references exist. An entry item consists of a pointer to the local object and a reference count. The reference count denotes the number of foreign heaps that have links to that object. The garbage collector of a heap treats all entry items as roots. For each heap, K0 also keeps a list of *exit items* for non-local objects to which the heap refers. An exit item contains a pointer to the entry item of the object to which it refers. At the end of a garbage collection cycle, unreferenced exit items are collected and the reference counts in the corresponding entry items are decremented. An entry item can be re-



claimed if its reference count reaches zero.

Write barriers are used to automatically create and update exit and entry items, as well as to maintain the heap reference invariants described previously. If a write barrier detects a reference that is legal, it will lookup and create the corresponding exit item for the remote object. In turn, the corresponding entry item in the foreign heap is updated. The same write barrier is used to prevent the passing of illegal cross-heap references. If the reference that would be created by a write is illegal, a segmentation violation error is thrown. The use of a write barrier is similar to the use of write checks in Omniware [2]. Although it may seem odd to use another protection mechanism (software fault isolation) in a type-safe system, the motivation is resource management, not memory safety.

Finally, to improve the use of the JVM's memory as a whole, K0 does not reserve non-overlapping, contiguous memory regions for each heap. Instead, memory accounting is done on a per-block basis. Small objects are stored in page-sized blocks, whereas larger objects are stored in dedicated blocks. Heaps receive new memory in blocks, and the garbage collector only reimburses a heap if it frees a whole block.

**CPU Management.** In traditional Java, each thread belongs to a thread group. Thread groups form a hierarchy in which each thread group has a parent group. The initial thread group is the root of the group hierarchy. K0 adapts the thread group classes such that all threads belonging to a process are contained in a subtree. Process threads cannot traverse this tree past the root of this subtree.

K0 combines the thread group hierarchy with CPU inheritance scheduling [23]. CPU inheritance scheduling is based on a directed yield primitive: a scheduler thread donates CPU time to a specific thread by yielding to it, which effectively schedules that thread. Since the receiver thread may in turn function as a scheduler thread, scheduler hierarchies can be built. Each non-root thread has an associated scheduler thread that is notified when that thread is runnable. A scheduler may use a timer to revoke its donation, which preempts a scheduled thread. Using CPU inheritance scheduling allows K0 to do two things. First, K0 can provide each process with its own scheduler that may implement any process-specific policy to schedule the threads in that process. Second, thread groups within processes may hierarchically schedule the threads belonging to them.

Each thread group in K0 is associated with a scheduler, which is an abstract Java class in K0. Different policies are implemented in different subclasses. At the root of the scheduling hierarchy, K0 uses a fixed priority policy to guarantee that the system heap garbage collector is given the highest priority. At the next level, a stride scheduler divides CPU time between processes. To pro-

vide compatibility with traditional Java scheduling, the root thread group of each process by default is associated with a fixed-priority scheduler that is a child of the stride scheduler.

#### 4.2.3 Implementation Status

We have prototyped a K0 kernel that is composed of a modified JVM that is based on Kaffe 1.0beta1. It is supplemented by classes in binary format from JavaSoft's JDK 1.1.5, and a package of privileged classes that replace part of the core java packages. K0 ran as a stand-alone kernel based on the OSKit [21] (a suite of components for building operating systems). Additionally, K0 ran in user-mode with libraries that simulate certain OSKit components such as interrupt handling and raw device access. We implemented separate heaps, as well as write barriers. Our initial prototype did not support separate garbage collection nor class garbage collection. The prototype supported CPU inheritance scheduling in the way described above, although it only supported schedulers implemented as native methods in C. We implemented four different policies: fixed-priority, rate-monotonic scheduling, lottery, and stride-scheduling.

We are currently working on a successor system called KaffeOS, which is based on a much improved base JVM, supports separate garbage collection, and will provide full resource reclamation.

#### 4.2.4 Summary

K0's design is oriented towards complete resource isolation between processes, with the secondary goal of allowing direct sharing. By giving each process a separate heap, many memory and CPU management resource issues become simpler. Sharing occurs through a shared system heap, and distributed garbage collection techniques are used to safely maintain sharing information.

### 4.3 Alta

Alta [44] is an extended Java Virtual Machine that provides a hierarchical process model and system API modeled after that provided by the Fluke microkernel. Fluke supports a *nested process model* [22], in which a process can manage all of the resources of child processes in much the same way that an operating system manages the resources of its processes. Memory management and CPU accounting are explicitly supported by the Alta system API. Higher-level services, such as network access and file systems, are managed by servers, with which applications communicate via IPC. Capabilities provide safe, cross-process references for communication.

Each process has its own root thread group, threads, and private copies of static member data. Per-process



memory accounting in Alta is comprehensive. For access control purposes, Alta expands the Fluke model by providing processes with the ability to control the classes used by a sub-process. Alta also extends the Java class model in that it allows a process to rename the classes that a subprocess sees. As a result, a process can interpose on all of a subprocess' interfaces.

The Alta virtual machine does not change any of the interfaces or semantics defined by the JVM specification. Existing Java applications, such as `javac` (the Java compiler), can run unmodified as processes within Alta.

#### 4.3.1 System Model

Communication in Alta is done through an IPC system that mimics the Fluke IPC system. Inter-process communication is based on a half-duplex, reversible, client-server connection between two threads (which may reside in different processes). Additionally, Alta IPC provides immediate notification to the client or server if the peer at the other end of the connection is terminated or disconnects.

Alta permits sibling processes to share objects directly. Objects can be shared by passing them through IPC. Sharing is only permitted for objects where the two processes have consistent views of the class name space. Enforcing this requirement efficiently requires that the classes involved are all final, i.e., that they cannot be subclassed. While this is somewhat restrictive, all of the primitive types — such as `byte[]` (an array of bytes) and `java.lang.String` — and many of the core Alta classes meet these requirements.

#### 4.3.2 Resource Management

The strongest feature of the Alta process model is the ability to “nest” processes: every process can manage child processes in the same way the system manages processes. Resource management in Alta is strictly hierarchical. Any process can create a child process and limit the memory allowance of that process.

**Memory Management.** Alta supports memory accounting through a simple allocator-pays scheme. The garbage collector credits the owning process when an object is eventually reclaimed. When a process is terminated, any existing objects are “promoted” into its parent's memory. Thus, it the responsibility of the parent process to make sure that cross-process references are not created if full memory reclamation is necessary upon child process termination. It is important to note that Alta enables a process to prevent child processes from passing Java object references through IPC.

Memory reclamation is simple if a process only passes references to its children. In the nested process model, when a process is terminated all of its child processes are necessarily terminated also. Therefore, refer-

ences that are passed to a process's children will become unused.

To support clean thread and process termination, Alta uses standard operating system implementation tricks to prevent the problem of threads terminated while executing critical system code, just like in K0. For example, to avoid stack overflows while executing system code, the entry layer will verify sufficient space is available on the current thread stack. This check is analogous to the standard technique of pre-allocating an adequate size stack for in-kernel execution in traditional operating systems. Additionally, Alta is structured to avoid explicit memory allocations within “kernel mode.” A system call can allocate objects before entering the kernel proper. Thus, all allocation effectively happens in “user mode.” Since the notion of the system code entry layer is explicit, some system calls (for example, `Thread.currentThread()`) never need enter the kernel.

**CPU Management.** Alta provides garbage collection as a system service. This leaves Alta open to denial-of-service attacks that generate large amounts of garbage—which will cause the garbage collector to run. Given the memory limits on processes, and limits on the CPU usage of a process, GC problems like this can be mitigated.

#### 4.3.3 Implementation Status

Alta's implementation is based on a JDK 1.0.2-equivalent JVM and libraries (Kaffe [49] version 0.9.2 and Kore [14] version 0.0.7, respectively). The bulk of the system is implemented entirely in Java. The internals of the VM were enhanced to support nested processes. A number of the core library classes were modified to use Alta primitives and to make class substitution more effective. In addition to `javac`, Alta supports simple applications that nest multiple children and control their class name spaces, along with a basic shell and other simple applications.

In terms of code sharing, a process in Alta is analogous to a statically linked binary in a traditional systems — each process has its own JIT'd version of a method. The Kaffe JIT could be modified to provide process-independent, sharable code, just as compilers can generate position-independent code for shared libraries. Like the version of Kaffe on which it is based, Alta does not yet support garbage collection of classes.

Alta does not implement CPU inheritance scheduling. Because Alta and K0 share a common code base, the CPU inheritance scheduling that is implemented in the K0 should be easy to migrate to Alta. In addition, like K0, Alta runs as a regular process on a traditional operating system and could be made to run on top of bare hardware using the OSKit.

#### 4.3.4 Summary

Alta implements the Fluke nested process model and API in a Java operating system. It demonstrates that the nested process model can provide Java processes with flexible control over resources. Because of the hierarchical nature of the model, direct sharing between siblings can be supported without resource reclamation problems.

#### 4.4 Performance Evaluation

We ran several microbenchmarks on our two prototype systems, Alta and K0, and a port of the J-Kernel to Kaffe to measure their baseline performance. These benchmarks demonstrate that no undue performance penalties are paid by “normal” Java code, in any of these systems, for supporting processes. In addition, they roughly compare the cost of IPC and Java process creation in all three systems.

The Alta, J-Kernel, and basic Kaffe tests were performed on a 300 MHz Intel Pentium II with 128MB of SDRAM. The system ran FreeBSD version 2.2.6, and was otherwise idle. The K0 tests were performed on the same machine, but K0 was linked to the OSKit and running without FreeBSD.

Table 1 shows the average time for a simple null instance method invocation, the average cost of allocating a `java.lang.Object`, the average overhead of creating and starting a `Thread` object, and the average cost of creating a `Throwable` object. All of the benchmarks were written to avoid invocation of the GC (intentional or unintentional) during timing. For K0 and Alta the benchmarks were run as the root task in the system. For the J-Kernel, the benchmarks were run as children of the J-Kernel `RootTask`, `cornell.slk.jkernel.std.Main`.

None of the systems significantly disrupts any of the basic features of the virtual machine. Previously published results about the J-Kernel [28] used Microsoft’s Java virtual machine, which is significantly faster than Kaffe. The Alta null thread test is significantly more expensive than the basic Kaffe test because Alta threads maintain additional per-thread state for IPC, process state, and blocking.

Table 2 measures the two critical costs of adding a process model to Java. The first column lists the overhead of creating a new process, measured from the time the parent creates the new process to the time at which the new process begins its main function. The Kaffe row lists the time required for Kaffe to fork and exec a new Kaffe process in FreeBSD. The J-Kernel supports a more limited notion of process—J-Kernel processes do not require an active thread—so the J-Kernel test simply creates a passive `Task` and seeds it with a simple initial object.

The subsequent columns of Table 2 show the time required for cross-task communication. Alta IPC is significantly slower because it is a rendezvous between two threads and uses ports, whereas J-Kernel IPC is simply cross-process method invocation. K0 IPC is implemented using a shared rendezvous object and is based directly on wait/notify. The IPC cost in K0 reflects its unoptimized thread package that is different than the thread package in the other JVMs.

Our performance results indicate that our systems need substantial optimization in order to realize the performance potential of language-based operating systems. The performance benefits from fine-grained sharing in software can be dominated by inefficiencies in the basic JVM implementation. As the difference to previously published J-Kernel results demonstrates, the future performance of Java systems will likely be spurred by advances in just-in-time compilation, which is orthogonal to the research issues we are exploring.

To analyze the implementation costs of our decision to build our own JVM, we examined each system in terms of useful lines of code (i.e., non-blank, non-comment lines of source). As a reference point, the original version of Kaffe v0.9.2 contains 10,000 lines of C, while Kaffe v1.0beta1 is comprised of just over 14,000 lines of C and 14,000 lines of Java. (Much of this increase is due to the move from JDK 1.0 to JDK 1.1.) Alta is comprised of 5,000 lines of Java and adds approximately 5,000 lines of C to Kaffe v0.9.2 (a significant fraction of this C code consists of features from later versions of Kaffe that we ported back to Kaffe v0.9.2). K0 adds approximately 1,000 lines of C code to the virtual machine and almost 2,000 lines of Java code to the basic libraries. The additional C code consisted of changes to the garbage collector to support K0’s separate heaps.

In comparison, the J-Kernel consists of approximately 9,000 lines of Java. Building the J-Kernel as a layer on top of a JVM was probably an easier implementation path than building a new JVM. The primary difficulty in building the J-Kernel probably lay in building the dynamic stub generator.

## 5 Related Work

Several lines of research are related to our work. First, the development of single-address-space operating systems — with protection provided by language or by hardware — is a direct antecedent of work in Java. Second, a great deal of research today is directed at building operating system services in Java.

### 5.1 Prior Research

A great deal of research has been done on hardware-based single-address-space operating systems. In

Virtual Machine	Method Invocation	Object Creation	Null Thread Test	Exception Creation
Kaffe 1.0beta1	0.16 $\mu$ s	1.9 $\mu$ s	480 $\mu$ s	12 $\mu$ s
K0	0.16 $\mu$ s	3.1 $\mu$ s	725 $\mu$ s	18 $\mu$ s
Alta	0.16 $\mu$ s	2.5 $\mu$ s	1030 $\mu$ s	15 $\mu$ s
Kaffe 0.10.0	0.17 $\mu$ s	1.8 $\mu$ s	470 $\mu$ s	10 $\mu$ s
J-Kernel	0.17 $\mu$ s	1.8 $\mu$ s	480 $\mu$ s	29 $\mu$ s

Table 1: Despite the fact that we have five distinct Java virtual machines based around different versions of the Kaffe virtual machine, base performance of the versions are not very different. The J-Kernel is run on Kaffe 0.10.0, because of deficiencies in object serialization in Kaffe 1.0beta1.

Virtual Machine	Process Creation	Null IPC	3-integer request	100-byte String request
Alta	120ms	90 $\mu$ s	109 $\mu$ s	138 $\mu$ s
K0	89ms	57 $\mu$ s	57 $\mu$ s	183 $\mu$ s
J-Kernel	235ms	2.7 $\mu$ s	2.7 $\mu$ s	27 $\mu$ s
Kaffe	300ms	N/A	N/A	N/A

Table 2: Process Tests. Note that numbers in the first column are reported in ms, while the other columns are reported in  $\mu$ s. Alta and K0 IPC is between separate threads while the J-Kernel IPC uses cross-process thread migration. The 3-integer request and 100-byte String request operations include the time to marshal and unmarshal the request. The J-Kernel uses object serialization to transmit a String while K0 and Alta use hand-coded String marshal and unmarshal code.

Opal [13], communication was accomplished by passing 256-bit capabilities among processes: a process could *attach* a memory segment to its address space so that it could address the memory segment directly. Because Opal was not based on a type-safe language, resource allocation and reclamation was coarse-grained, and based on reference counting of segments.

Many research projects have explored operating systems issues within the context of programming languages. For example, Argus [33] and Clouds [16] explored the use of transactions within distributed programming languages. Other important systems that studied issues of distribution include Eden [3], Emerald [11], and Amber [12]. These systems explored the concepts underlying object migration, but did not investigate resource management.

Language-based operating systems have existed for many years. Most of them were not designed to protect against malicious users, although a number of them support strong security features. None of them, however, provide strong resource controls. Pilot [38] and Cedar [43] were two of the earliest language-based systems. Their development at Xerox PARC predates a flurry of research in the 1990's on such systems.

Oberon [51] has many of Java's features, such as garbage collection, object-orientation, strong type-checking, and dynamic binding. Unlike Java, Oberon is a non-preemptive, single-threaded system. Background tasks like the garbage collector are implemented as calls to procedures, where "interruption" can only occur between top-level procedure calls.

A related project, Juice [24] provides an execution environment for downloaded Oberon code (just as a JVM provides an execution environment for Java). Juice is a virtual machine that executes "binaries" in its own portable format: it compiles them to native code during loading, and executes the native code directly. The advantage of Juice is that its portable format is faster to decode and easier to compile than Java's bytecode format.

SPIN [9] is an operating system kernel that lets applications load extensions written in Modula-3 that can extend or specialize the kernel. As with Java, the type safety of Modula-3 ensures memory safety. SPIN supports dynamic interposition on names, so that extensions can have different name spaces.

Inferno [19], an OS for building distributed services, has its own virtual machine called Dis and its own programming language called Limbo. Inferno is a small system that has been ported to many architectures: it has been designed to run in resource-limited environments, such as set-top boxes. In order to minimize garbage collection pauses, Inferno uses reference counting to reclaim memory, avoiding a number of accounting issues related to garbage collection in an operating system.

VINO is a software-based (but not language-based) extensible system [40] that addresses resource control issues by wrapping kernel extensions within transactions. When an extension exceeds its resource limits, it can be safely aborted (even if it holds kernel locks), and its resources can be recovered.



## 5.2 Java-Based Research

Besides Alta, K0, and the J-Kernel, a number of other research systems have explored (or are exploring) the problem of supporting processes in Java.

Balfanz and Gong [6] describe a multi-processing JVM developed to explore the security architecture ramifications of protecting applications from each other, as opposed to just protecting the system from applications. They identify several areas of the JDK that assume a single-application model, and propose extensions to the JDK to allow multiple applications and to provide inter-application security. The focus of their multi-processing JVM is to explore the applicability of the JDK security model to multi-processing, and they rely on the existing, limited JDK infrastructure for resource control.

IBM [18] released a JVM for its OS/390 family of systems that is targeted towards server applications such as Enterprise Java Beans. Their system puts each transaction into a separate worker JVM that initialize from and execute out of a shared heap. This shared heap holds those classes and objects that are expected to survive a transaction. Worker JVMs that leave no resources behind can be reused for multiple transactions. If a transaction does leave resources behind, the worker JVM process is terminated and the OS is used to free those resources. IBM's motivation for providing a quasi-process model in Java are faster startup times attributable to the savings in class loading and processing, which increases transaction throughput. However, they do not consider the case of malicious and uncooperative applications because there is no control over what data individual applications can store on the shared heap. In addition, the shared heap is not garbage collected.

One approach to resource control is to dedicate an entire machine to the execution of client code. For instance, AT&T's "Java Playground" [34] and Digitivity's "CAGE" Applet Management System [17] define special Java applet execution models that require applets to run on dedicated, specially protected hosts. This execution model imposes extremely rigid limits on mobile code, by quarantining applets on isolated hosts. As a result, richer access is completely disallowed. Although the above-mentioned systems guarantee the integrity of the JVM, they do not provide any inter-applet guarantees beyond that offered by the underlying "stock" JDK. These systems are similar to Kimera [41], which uses dedicated servers to protect critical virtual machine resources (e.g., the bytecode verifier) but not to protect applications from each other.

Luna [29] is a recent system from one of the J-Kernel developers. Luna extends the Java language and runtime with explicit, revocable remote pointers. Remote pointers can be dynamically revoked, and processes can safely share fine-grained data without compromising

type-safety.

Sun's original JavaOS [42] was a standalone OS written almost entirely in Java. It is described as a first-class OS for Java applications, but appears to provide a single JVM with little separation between applications. It is being replaced by a new implementation termed "JavaOS for Business" that also only runs Java applications. "JavaOS for Consumers" is built on the Chorus microkernel OS [39] in order to achieve real-time properties needed in embedded systems. Both of these systems require a separate JVM for each Java application, and all run in supervisor mode.

Joust [27], a JVM integrated into the Scout operating system [35], provides control over CPU time and network bandwidth. To do so, it uses Scout's path abstraction. However, Joust does not provide memory limits.

The Open Group's Conversant system [7] is yet another project that modifies a JVM to provide processes. It provides each process with a separate address range (within a single Mach task), a separate heap, and a separate garbage collection thread. Conversant does not support sharing between processes, unlike our systems and the J-Kernel. Its threads are native Mach threads that support POSIX real-time semantics. Conversant provides some real-time services. Another real-time system, PERC [36], extends Java to support real-time performance guarantees. The PERC system analyzes Java bytecodes to determine memory requirements and worst-case execution time, and feeds that information to a real-time scheduler.

## 6 Conclusions

In order to support multiple applications, a Java operating system must control computational resources. The major technical challenges that must be addressed in building such a system are managing memory and CPU usage for shared code. Some of these challenges can be dealt with by adapting techniques used in conventional systems to language-based systems. Other challenges can be dealt with by adapting language technology, such as garbage collection, to fit into an operating system framework.

We have described two prototype Java operating systems that are being built at Utah: Alta and K0. These two prototypes and Cornell's J-Kernel illustrate tradeoffs that can be made in terms of system structure, resource management, and implementation strategies. We have shown that many design issues from conventional operating systems resurface in the structural design of Java operating systems. Java operating systems can be built with monolithic designs, as K0; or they can be built with microkernel designs, as Alta or the J-Kernel. Finally, we have shown how garbage collection techniques can be used to



support resource management for Java processes.

## Acknowledgments

We thank Kristin Wright, Stephen Clawson, and James Simister for their efforts in helping us with the results. We thank Eric Eide for his great help in editing and improving the presentation of this material, and Massimiliano Poletto for his comments on drafts of this paper. We thank Chris Hawblitzel for his clarifications of how the J-Kernel works. Finally, we thank the Flux group for their work in making the OSKit, which was used in some of this work.

## References

- [1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proc. of the Summer 1986 USENIX Conf.*, pages 93–112, June 1986.
- [2] A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In *Proc. ACM SIGPLAN '96 Conf. on Programming Language Design and Implementation (PLDI)*, pages 127–136, May 1996.
- [3] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, Jan. 1985.
- [4] The Java Apache project. <http://java.apache.org>, Apr. 2000.
- [5] G. V. Back and W. C. Hsieh. Drawing the red line in Java. In *Proc. of the Seventh Workshop on Hot Topics in Operating Systems*, pages 116–121, Rio Rico, AZ, Mar. 1999. IEEE Computer Society.
- [6] D. Balfanz and L. Gong. Experience with secure multiprocessing in Java. In *Proc. of the Eighteenth International Conf. on Distributed Computing Systems*, May 1998.
- [7] P. Bernadat, L. Feeney, D. Lambright, and F. Travostino. Java sandboxes meet service guarantees: Secure partitioning of CPU and memory. Technical Report TOGRI-TR9805, The Open Group Research Institute, June 1998.
- [8] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, Feb. 1990.
- [9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, Dec. 1995.
- [10] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1), Feb. 1984.
- [11] A. P. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Trans. on Software Engineering*, SE-13(1):65–76, 1987.
- [12] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 147–158, December 1989.
- [13] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, 1994.
- [14] G. Clements and G. Morrison. Kore — an implementation of the Java(tm) core class libraries. <ftp://sensei.co.uk/misc/kore.tar.gz> OR <http://www.cs.utah.edu/projects/flux/java/kore/>.
- [15] G. Czajkowski, C.-C. Chang, C. Hawblitzel, D. Hu, and T. von Eicken. Resource management for extensible internet servers. In *Proceedings of the 8th ACM SIGOPS European Workshop*, Sintra, Portugal, Sept. 1998.
- [16] P. Dasgupta et al. The design and implementation of the Clouds distributed operating system. *Computing Systems*, 3(1), Winter 1990.
- [17] Digitivity Corp. Digitivity CAGE, 1997. <http://www.digitivity.com/overview.html>.
- [18] D. Dillenberger, R. Bordawekar, C. W. Clark, D. Durand, D. Emmes, O. Gohda, S. Howard, M. F. Oliver, F. Samuel, and R. W. St. John. Building a Java virtual machine for server applications: The JVM on OS/390. *IBM Systems Journal*, 39(1):194–210, 2000. Reprint Order No. G321–5723.
- [19] S. Dorward, R. Pike, D. L. Presotto, D. Ritchie, H. Trickey, and P. Winterbottom. Inferno. In *Proceedings of the 42nd IEEE Computer Society International Conference*, San Jose, CA, February 1997.
- [20] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 261–275, Seattle, WA, Oct. 1996.
- [21] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for OS and language research. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, St. Malo, France, Oct. 1997.
- [22] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 137–151. USENIX Association, Oct. 1996.
- [23] B. Ford and S. Susarla. CPU inheritance scheduling. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 91–105, Seattle, WA, Oct. 1996. USENIX Association.

- [24] M. Franz. Beyond Java: An infrastructure for high-performance mobile code on the World Wide Web. In S. Lobodzinski and I. Tomek, editors, *Proceedings of WebNet '97*, pages 33–38, October 1997.
- [25] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2. In *Proc. of USENIX Symposium on Internet Technologies and Systems*, pages 103–112, Monterey, CA, Dec. 1997. USENIX.
- [26] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1996.
- [27] J. H. Hartman et al. Joust: A platform for communication-oriented liquid software. Technical Report 97–16, Univ. of Arizona, CS Dept., Dec. 1997.
- [28] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *Proc. of the USENIX 1998 Annual Technical Conf.*, pages 259–270, New Orleans, LA, June 1998.
- [29] C. Hawblitzel and T. von Eicken. Tasks and revocation for Java (or, hey! you got your operating system in my language!). Describes Luna, Nov. 1999.
- [30] J. Kiniry and D. Zimmerman. Special feature: A hands-on look at Java mobile agents. *IEEE Internet Computing*, 1(4), July/August 1997.
- [31] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proc. of OOPSLA '98*, Vancouver, BC, Oct. 1998. To appear.
- [32] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Jan. 1997.
- [33] B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.
- [34] D. Malkhi, M. K. Reiter, and A. D. Rubin. Secure execution of Java applets using a remote playground. In *Proc. of the 1998 IEEE Symposium on Security and Privacy*, pages 40–51, Oakland, CA, May 1998.
- [35] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 153–167, Seattle, WA, Oct. 1996. USENIX Association.
- [36] K. Nilsen. Java for real-time. *Real-Time Systems Journal*, 11(2), 1996.
- [37] D. Plainfossé and M. Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of the 1995 International Workshop on Memory Management*, Kinross, Scotland, Sept. 1995.
- [38] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, 1980.
- [39] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. The Chorus distributed operating system. *Computing Systems*, 1(4):287–338, Dec. 1989.
- [40] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, WA, Oct. 1996. USENIX Association.
- [41] E. Sirer, R. Grimm, A. Gregory, and B. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *Proceedings of the 17th Symposium on Operating Systems Principles*, pages 202–216, Kiawah Island Resort, SC, Dec. 1999.
- [42] Sun Microsystems, Inc. JavaOS: A standalone Java environment, Feb. 1997. <http://www.javasoft.com/products/javaos/javaos.white.html>.
- [43] D. C. Swinehart, P. T. Zellweger, R. J. Beach, and R. B. Hagmann. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, Oct. 1986.
- [44] P. Tullmann and J. Lepreau. Nested Java processes: OS structure for mobile code. In *Proc. of the Eighth ACM SIGOPS European Workshop*, pages 111–117, Sintra, Portugal, Sept. 1998.
- [45] T. von Eicken, C.-C. Chang, G. Czajkowski, C. Hawblitzel, and D. Hu. J-Kernel. Source code available at <http://www.cs.cornell.edu/slk/jk-0.91/doc/Default.html>.
- [46] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and S. Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, Feb. 1992.
- [47] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 116–128, Oct. 1997.
- [48] D. J. Wetherall, J. Guttag, and D. L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *Proceedings of IEEE OPENARCH '98*, San Francisco, CA, April 1998.
- [49] T. Wilkinson. Kaffe—a virtual machine to compile and interpret Java bytecodes. <http://www.transvirtual.com/kaffe.html>.
- [50] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the 1992 International Workshop on Memory Management*, St. Malo, France, Sept. 1992.
- [51] N. Wirth and J. Gutknecht. *Project Oberon*. ACM Press, New York, NY, 1992.

# Signaled Receiver Processing

José Brustoloni, Eran Gabber, Abraham Silberschatz and Amit Singh  
*Information Sciences Research Center*  
*Lucent Technologies — Bell Laboratories*  
600 Mountain Avenue, Murray Hill, NJ 07974, USA  
{jcb, eran, avi, amitsingh}@research.bell-labs.com

## Abstract

*Protocol processing of received packets in BSD Unix is interrupt-driven and may cause scheduling anomalies that are unacceptable in systems that provide quality of service (QoS) guarantees. We propose an alternative mechanism, Signaled Receiver Processing (SRP), that generates a signal to the receiving process when a packet arrives. The default action of this signal is to perform protocol processing asynchronously. However, a receiving process may catch, block, or ignore the signal and defer protocol processing until a subsequent receive call. In any case, protocol processing occurs in the context of the receiving process and is correctly charged. Therefore, SRP allows the system to enforce and honor QoS guarantees. SRP offers several advantages over Lazy Receiver Processing (LRP), a previous solution to BSD's scheduling anomalies: SRP is easily portable to systems that support neither kernel threads nor Resource Containers (e.g., FreeBSD); gives applications control over the scheduling of protocol processing; uses a demultiplexing strategy that is appropriate for both hosts and gateways; and easily enables real-time or proportional-share scheduling.*

## 1 Introduction

Many Internet protocols, including TCP and IP, first appeared on the BSD Unix operating system [19]. BSD implementations were widely circulated, evaluated, and debugged both in industry and academia, thus influencing many other implementations [23]. Such influence promoted interoperability, but also disseminated artifacts that are neither desirable nor part of the protocol standards.

One of these artifacts is that, in BSD-inspired im-

plementations, protocol processing of received packets typically occurs in the context of a software interrupt, before the system demultiplexes packets to socket receive queues. In many implementations, this processing is charged to whatever application was interrupted, even if the latter is unrelated to the packets. In other implementations, this processing is not charged at all. In either case, protocol processing may thus cause scheduling anomalies; the system cannot enforce CPU allocations and therefore cannot provide quality of service (QoS) guarantees to applications. Furthermore, BSD's scheme always prioritizes protocol processing of a received packet over application processing, even if the respective socket receive queue is full and therefore the system will have to drop the packet. In fact, under high receive loads, the system may waste all CPU time processing packets that will have to be dropped (a phenomenon known as *receive livelock* [20]): The system gives applications no CPU time and therefore applications cannot empty the receive queues. Receive livelock can be exploited in denial of service attacks.

BSD's scheduling anomalies can be avoided by an alternative scheme, Lazy Receiver Processing (LRP) [14]. LRP's techniques are transparent to applications and do not violate protocol standards. LRP combines several mechanisms to guarantee that resources used in a packet's protocol processing are charged to the application that will receive that packet. LRP uses *early demultiplexing*, i.e., demultiplexes packets to the respective receiving applications *before* protocol processing. In the case of UDP packets, LRP always processes protocols *synchronously*, i.e., when the receiving application issues a receive call. On the other hand, LRP always processes TCP packets *asynchronously*, i.e., when packets arrive. For correct resource accounting, LRP may associate with each process an extra kernel thread that asynchronously processes incom-



ing TCP packets for the respective process, and has its resource utilization charged to the process [14]. Alternatively, LRP may process all incoming TCP packets in a single process, and use a Resource Containers facility to charge resource usage to the Containers of the respective receiving applications [1].<sup>1</sup>

Unfortunately, LRP may also present significant difficulties. First, many operating systems support neither kernel threads (e.g., FreeBSD) nor Resource Containers (e.g., most existing systems). Therefore, it can be difficult to port LRP to such systems. Second, LRP's UDP processing is always synchronous, whereas LRP's TCP processing is always asynchronous and shares resources equally with the receiving application. However, for some applications, different protocol scheduling or resource apportionment may be preferable. Third, LRP and Resource Containers were designed for *hosts* (as acknowledged by the reference to "server systems" in the titles of the respective papers [14, 1]). However, scheduling and resource management in *gateways* is becoming as important as in *hosts*. Gateways no longer simply forward packets; they increasingly also need to run applications such as routing protocols, network management [10], firewalling, Network Address Translation (NAT) [15], load balancing [22], reservation protocols [4], or billing [12]. Extensible routers [21] and active networks [9] suggest a number of other ways in which it may be advantageous to run application-specific code on gateways.

LRP's use in these modern gateways faces two problems. First, LRP's early demultiplexing does not provide the required flexibility. In gateways, each packet may need to be processed not by a single receiving application, but by a variable series of applications, each of which may modify the packet's header and affect what other applications need to process the packet. Second, LRP and Resource Containers were described and evaluated in detail only in conjunction with time-sharing scheduling. However, this type of scheduling may be inadequate for gateways. For example, it would be improper to penalize IP forwarding according to its CPU usage, as a typical time-sharing scheduler would. On the other hand, giving IP forwarding a "real-time" priority

<sup>1</sup>In many operating systems, including FreeBSD, the notions of resource principal and protection domain coincide in the process abstraction. Resource Containers are a proposal to separate these notions, making resource management more flexible. For example, a given client's resource consumption may be represented by a single Resource Container. In this case, resources used by different servers on behalf of the client may be charged to that client's Resource Container.

may also be inadequate, because it could lead to the starvation of time-sharing or other lower-priority applications (e.g., in FreeBSD 3.0, real-time priorities are fixed and are always higher than time-sharing priorities).

This paper contributes a new scheme, Signaled Receiver Processing (SRP), that overcomes both BSD's and LRP's mentioned shortcomings. When an incoming packet is demultiplexed to a given process, SRP *signals* that process. The *default* action on such signal is to perform protocol processing asynchronously. However, a process may choose to synchronize protocol processing by *catching*, *blocking*, or *ignoring* SRP's signals. In the latter cases, protocol processing is deferred until a later receive call. In all cases, protocol processing occurs in the context of and is charged to the receiving process.

SRP has several advantages over LRP. First, SRP uses signals and not kernel threads nor Resource Containers. Therefore, SRP can be easily ported to most existing systems, including FreeBSD. Second, SRP gives applications considerable control over the scheduling and resource apportionment of protocol processing. For example, an application may catch SRP's signals to control the time spent doing protocol processing; block SRP's signals to avoid interruptions while processing some urgent event; or ignore SRP's signals to make TCP processing synchronous. Synchronous TCP processing can improve memory locality. Such control is not possible in LRP because LRP was designed to be transparent to applications. Third, SRP supports modern gateways. SRP uses a multi-stage demultiplexing function that, unlike LRP's simple early demultiplexing, allows packets to be examined and modified by a multiple and possibly variable series of applications. Moreover, SRP supports proportional-share scheduling. In a gateway, proportional-share scheduling can guarantee to each application (e.g., IP forwarding, load balancing, or billing) a minimum share of the CPU, without penalties for usage and without starvation of other applications. We implemented SRP as part of Eclipse/BSD, a new operating system that is derived from FreeBSD and that provides QoS guarantees via proportional-share scheduling of each system resource, including CPU, disk, and network output link bandwidth [7].

The rest of this paper is organized as follows. Sections 2 and 3 describe in greater detail how BSD and LRP process received packets, respectively. Section 4 reports the difficulties we encountered when



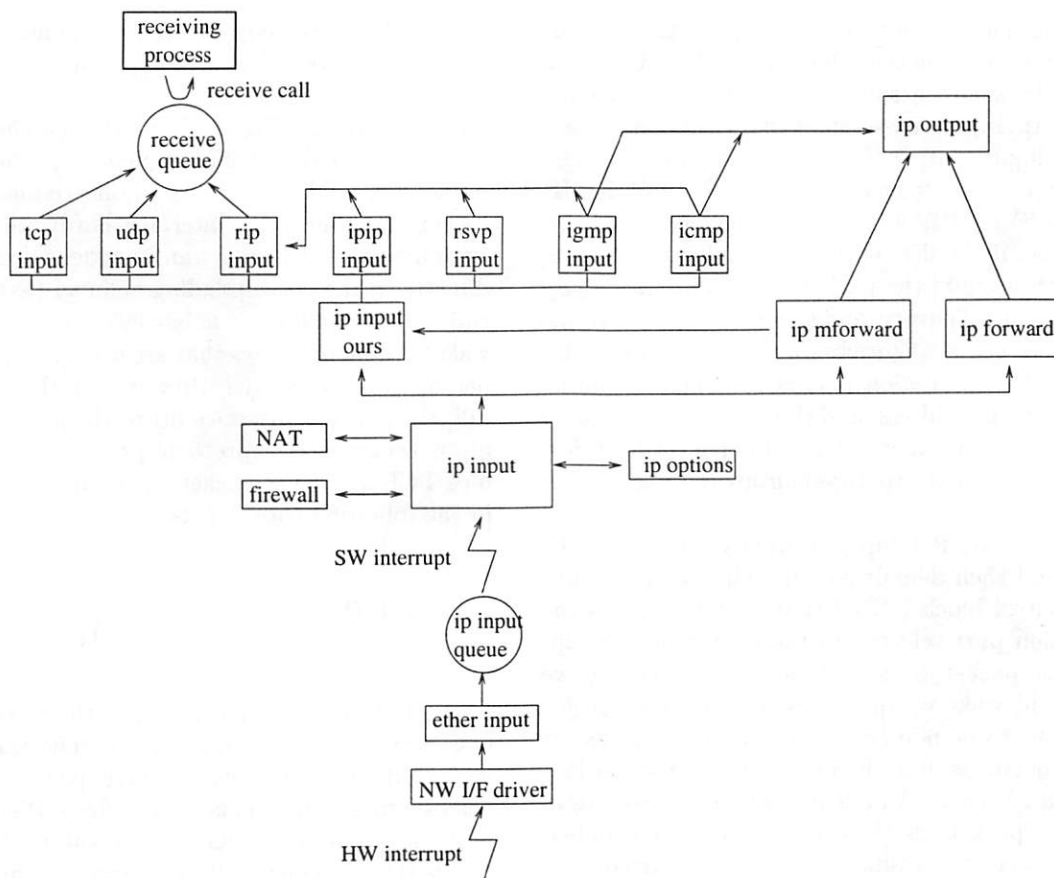


Figure 1: In FreeBSD, protocol processing of received packets occurs in the context of a hardware or software interrupt at priority higher than that of any application, and is charged to whatever process was interrupted.

porting LRP to FreeBSD for the implementation of Eclipse/BSD. Those difficulties led to the design of SRP, described in Section 5. Experiments in Section 6 demonstrate that, like LRP, SRP prevents receive livelock. However, the experiments also show that SRP supports Eclipse/BSD's QoS guarantees, that such support is desirable for gateway functionality, such as IP forwarding, and that SRP, unlike LRP, allows any application to control the scheduling of protocol processing, e.g. to achieve better memory locality. Section 7 discusses related work, and Section 8 concludes.

## 2 BSD receiver processing

This section discusses how FreeBSD processes IP packets received from an Ethernet. This discussion is representative also of protocol processing in other derivatives of BSD and for other protocol families

and networks.

As shown in Figure 1, packet arrival causes a hardware interrupt that transfers CPU control to a network interface driver. The driver retrieves the packet from the network interface hardware, prepares the hardware for receiving a future packet, and passes the received packet to the `ether_input` routine. `ether_input` places the packet in IP's input queue without demultiplexing: All IP packets go in the same queue. `ether_input` then issues a network software interrupt. This software interrupt has priority higher than that of any application, but lower than that of the hardware interrupt.

FreeBSD handles the network software interrupt by dequeuing each packet from IP's input queue and calling the `ip_input` routine. `ip_input` checksums the packet's IP header and submits the packet to preliminary processing: firewalling and/or NAT, if configured in the system, and IP options, if present in the packet header. This preliminary processing

may drop, modify, or forward the packet. `ip_input` then checks the packet's destination IP address. If that address is the same as one of the host's addresses, `ip_input` reassembles the packet and passes it to the input routine of the higher-layer protocol selected in the packet header (i.e., TCP, UDP, ICMP, IGMP, RSVP, IPIP, or, in remaining cases, raw IP). Otherwise, if the destination is a multicast address, `ip_input` submits the packet to a higher-layer protocol, for local delivery, and to multicast forwarding, if the system is configured as a multicast router. Finally, if the destination matches neither the host's nor a multicast address, and the system is configured as a gateway, `ip_input` submits the packet to IP forwarding; otherwise, `ip_input` drops the packet.

TCP's and UDP's input routines checksum the packet and then demultiplex it. They find the protocol control block (PCB) that corresponds to the destination port selected in the packet header, append the packet to the respective socket receive queue, and wake up processes that are waiting for that queue to be non-empty. However, if the socket receive queue is full, FreeBSD drops the packet. Note that, because demultiplexing occurs so late in FreeBSD, packets destined to the host are dropped *after* protocol processing has already occurred.

Note also that, in FreeBSD, protocol processing of a received packet is asynchronous relative to the respective receiving process. On receive calls, the receiving process checks the socket receive queue. If the queue is empty, the process sleeps; otherwise, the process dequeues the data and copies it out to application buffers.

However, processes only get a chance to run if the receive load is not so high that all CPU time is spent processing network hardware or software interrupts (receive livelock). Moreover, even at moderate receive loads, process scheduling may be disturbed by the fact that the CPU time spent processing network interrupts is charged to whatever process was interrupted, even if that process is unrelated to the received packets.

### 3 LRP

Although popular, BSD's scheme for processing received packets can cause scheduling anomalies, as discussed in the previous section. LRP [14] has been

proposed as a remedy to such anomalies. This section reviews how LRP achieves that.

As illustrated in Figure 2, LRP uses *channels* instead of a single IP input queue. A channel is a packet queue; LRP associates one channel to each socket. The network interface hardware or driver examines packet headers and enqueues each packet directly in the corresponding channel (early demultiplexing). Following a hardware interrupt, LRP wakes up the processes that are waiting for the channel to be non-empty. However, if the channel is full, the network interface drops the packet immediately, *before* further protocol processing. LRP handles TCP and UDP packets differently, as discussed in the following subsections.

#### 3.1 UDP

In the UDP case, on receive calls, the receiving process performs the following loop while there is not enough data in the socket receive queue: While the corresponding channel is empty, sleep; then dequeue each packet from the channel and submit the packet to `ip_input`, which calls `udp_input`, which finally enqueues the packet in the socket receive queue. The receiving process then dequeues the data from the socket receive queue and copies it out to application buffers. Therefore, for UDP, LRP is *synchronous* relative to the receiving process's receive calls.

#### 3.2 TCP

Unlike the UDP case, in the TCP case, LRP is *asynchronous* relative to the receiving process. LRP was designed to be completely transparent to applications and, in some applications, performing TCP processing synchronously relative to application receive calls could cause large or variable delays in TCP acknowledgments, adversely affecting throughput. In order to process TCP asynchronously without resorting to software interrupts, LRP may associate with each process an extra kernel thread that is scheduled at the process's priority and has its resource utilization charged to the process, as shown in Figure 2. This kernel thread continuously performs the following loop: While the process's TCP channels are empty, sleep; then dequeue each packet from a non-empty TCP channel and submit the packet to `ip_input`, which calls `tcp_input`, which finally en-

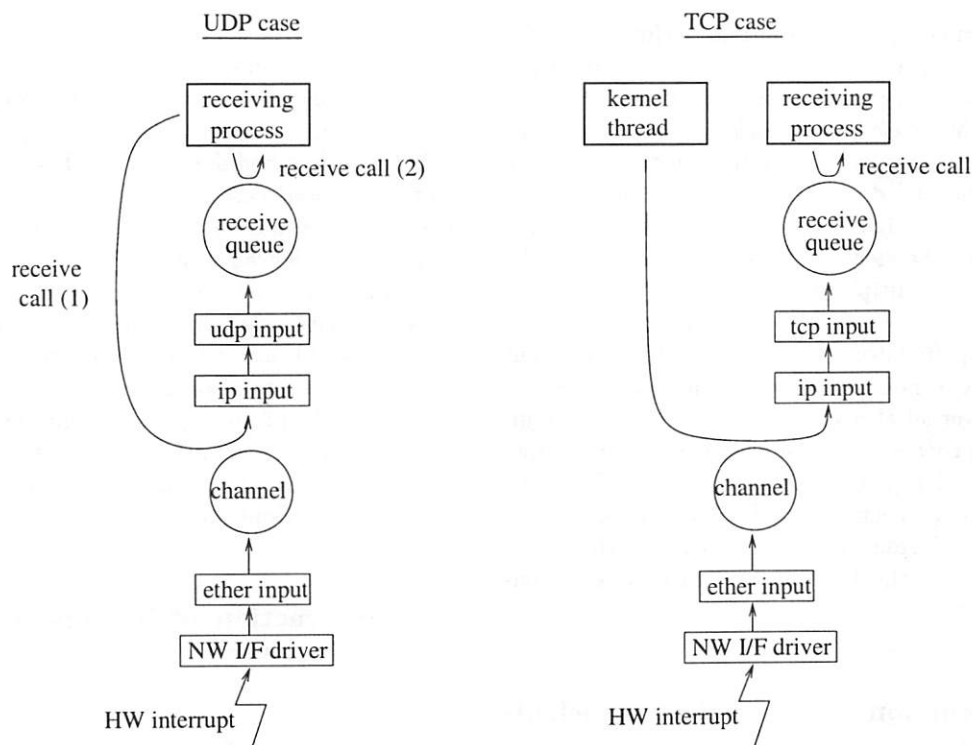


Figure 2: LRP processes received UDP packets in the context of the receiving process, when the latter issues a receive call. In the TCP case, LRP may use a kernel thread scheduled at the same priority as the receiving process.

queues the packet in the respective socket receive queue.

Instead of kernel threads, LRP may use a single process for similarly handling packets from *all* TCP channels in the system. In this case, LRP uses a Resource Containers facility to charge the resources used in processing each packet to the Container of the respective receiving application [1].

In either case, LRP handles TCP receive calls similarly to BSD: the receiving process simply checks the socket receive queue and, if the queue is empty, sleeps; otherwise, the process dequeues the data and copies it out to application buffers.

## 4 LRP's difficulties, shortcomings, and open questions

LRP was initially developed for a workstation time-sharing operating system, SunOS. However, LRP's ability to prevent certain networking-

related scheduling anomalies caught our attention when we were modifying FreeBSD to implement Eclipse/BSD [7]. Eclipse/BSD is a new operating system that provides QoS guarantees using proportional-share scheduling. Eclipse/BSD is intended for both hosts and gateways. Our initial intention was to use LRP, but we ran into the problems reported in this section.

Our most obvious difficulty was that FreeBSD supports neither kernel threads nor Resource Containers, which are used in LRP's TCP processing. We could have chosen to build either piece of infrastructure, but decided against that because (1) the effort required would be nontrivial, and (2) we found that the redesign described in the next section (SRP) both leads to an easier implementation and solves several shortcomings and open questions in LRP, as discussed in the following subsections.

### 4.1 Inflexibility

Although LRP is an architecture with several implementation options, it is unnecessarily inflexible.

First, implementors may choose to perform early demultiplexing either in hardware (network interface card) or in software (network interface driver). However, in either case, each packet is demultiplexed only once, directly to the application that will consume the packet's data. These options do not support gateways, where applications like NAT may modify packet headers, and packets may need to be demultiplexed multiple times.

Second, implementors may choose to implement LRP's asynchronous TCP processing using, e.g., an additional kernel thread per process, or a system-wide TCP process and Resource Containers. However, LRP's TCP processing is always asynchronous. In some cases, synchronous TCP processing could give better performance, e.g. because of better memory locality, but the LRP architecture does not enable such option.

#### 4.2 Interaction with real-time schedulers

The LRP and Resource Container papers [14, 1] suggest that the respective techniques can be used in conjunction with real-time or proportional-share schedulers, but do not satisfactorily explain how to achieve that. Those papers describe in reasonable detail only how LRP and Resource Containers are used in conjunction with time-sharing schedulers. The latter schedulers typically assign to each process a dynamic priority based on an average of the process's CPU usage [19]; no process has a fixed priority or CPU share.

Consider, in contrast, a real-time scheduler and processes that have fixed priorities. If LRP's asynchronous TCP processing is implemented with kernel threads, what should be the priority of those threads? By analogy to the time-sharing case, a TCP thread's priority would be the same as that of the respective process. However, in this case, process latencies and cumulative service [5] may suffer: Process events cannot preempt the TCP thread and may be delayed until the end of the latter's CPU quota.

On the other hand, if LRP is implemented with a single TCP process and Resource Containers, what should be the priority of the TCP process? Resource Containers calculate each thread's priority based on the thread's *scheduler binding*, i.e., the set of Con-

tainers recently served by the thread. In order to enable asynchronous TCP processing for all Containers, it appears that the TCP process's priority should be no less than the maximum priority of Containers in its scheduler binding. Therefore, similarly to the previous case (TCP kernel threads), application latencies and cumulative service may suffer. To reduce such degradation and prevent priority inversions and other scheduling anomalies, the TCP process's priority would have to be recomputed and the system would need to be rescheduled each time (1) the TCP process performs a *resource binding*, i.e., makes a call informing what Container it is about to serve, or (2) a packet arrives for a previously idle Container. These modifications may increase the Resource Container overhead considerably.

#### 4.3 Interaction with proportional-share schedulers

The interaction of LRP with proportional-share schedulers similarly raises a number of questions. If LRP's asynchronous TCP processing is implemented with kernel threads, what CPU shares should those threads have? By analogy with the time-sharing case, (1) a TCP thread should have the same share as that of the respective process, and (2) if a TCP thread does not fully utilize its allocation, the respective process should be given the excess allocation (and vice-versa). To meet these requirements, LRP would need a *hierarchical* (not a *flat*) proportional-share scheduler. In a hierarchical proportional-share scheduler, LRP can take a CPU share and split it into subordinate equal shares for a process and the respective TCP thread [7]. Flat proportional-share schedulers, however, do not allow the hierarchical subdivision of shares, and divide excess allocations among *all* processes or threads, even if they are unrelated. Thus, LRP cannot prevent scheduling anomalies if it is used with such schedulers.

On the other hand, if LRP is implemented with a single TCP process and Resource Containers, how should scheduling be performed? In the time-sharing case, to avoid excessive context switching, systems with Resource Containers schedule *threads*, not *Containers*. A thread is scheduled based on a priority computed according the thread's scheduler binding. It is unclear what the analogous construct would be in the hierarchical proportional-share case. In particular, what would be the map-



ping between the hierarchy and shares of, on the one hand, Containers, and on the other hand, the threads that may dynamically serve those Containers? Proportional-share schedulers often compute fairly elaborate virtual time or virtual work functions in order to approximate GPS (Generalized Processor Sharing) scheduling. These functions may depend not only on shares, but also on requests' start and finish times [3, 2, 17, 7]. How would virtual time or work in the hierarchical Container space map to equivalents in the thread space? Furthermore, Resource Containers require each thread to do its own scheduling of requests from different Containers served by the thread. In the time-sharing case, thread-level scheduling is according to Container priority [1]. But in a proportional-share case, how would thread-level Container scheduling be integrated with system-level thread scheduling so as to approximate a global hierarchical proportional-share scheduling of Containers?<sup>2</sup>

## 5 SRP

As discussed in the previous section, the use of LRP in a system based on FreeBSD and with proportional-share scheduling, such as Eclipse/BSD, is fraught with difficulties and open problems. To circumvent those problems, we designed the alternative solution, SRP, described in this section. We discuss in the following subsections SRP's protocol organization, packet demultiplexing, packet notification, and protocol scheduling. Finally, we summarize the advantages of our approach.

### 5.1 Protocol organization

This subsection gives an overview of how SRP organizes the protocol processing of received packets.

SRP does not require modifications in the network interface hardware or driver. As illustrated in Figure 3, packet arrival causes a network hardware

<sup>2</sup>The initial LRP prototype [14] used a system-wide TCP process. However, its reported results for TCP show only immunity to receive livelock, not correct resource accounting (the RPC results use UDP, which is always synchronous, and demonstrate the performance benefits of improving memory locality) [14]. Note that LRP's later implementation in the Resource Container prototype used an additional kernel thread per process for TCP processing, instead of a system-wide TCP process [1].

interrupt and transfers CPU control to the network interface driver, which passes the packet to `ether_input`.

In order to accommodate gateway functionality, such as firewalling and NAT, SRP organizes protocol processing in *stages*, where each stage comprises one or more protocol functions. Stages can be *preliminary* (including the `ether_input`, firewalling, NAT, and IP option handling stages) or *final* (including the end-application, ICMP, IGMP, RSVP, IPIP, raw IP, multicast, and IP forwarding stages). Preliminary stages invoke SRP's *next stage submit* (NSS) function to submit a packet to the respective next stage. Final stages are those that include IP and higher-layer protocols necessary to give a final disposition to each packet. The end-application stage, for example, includes IP, TCP, and UDP, and enqueues the packet in the corresponding socket receive queue.

Only the `ether_input` stage runs at interrupt level. The end-application stage runs in the context of the respective receiving application. All other stages run in the context of system processes with CPU guarantees from Eclipse/BSD. In the current SRP implementation, all protocol processing occurs inside the kernel. With minor SRP modifications, however, user-level processing would also be possible.

### 5.2 Packet demultiplexing and buffering

This subsection discusses how SRP demultiplexes and buffers incoming packets.

NSS is the central component in SRP's demultiplexing. NSS uses SRP's *multi-stage early demultiplex* (MED) function. MED returns a pointer to the PCB of the next stage to which a packet should be submitted, based on current stage and packet header. MED caches the PCB pointer in a field in the packet's first buffer, so that later, for example, TCP and UDP do not have to again look up the packet's PCB. Each PCB points to a socket, which in turn points to an *unprocessed input queue* (UIQ), to an *input function*, and to a list of *owner processes*, which are the processes that have the socket open. In order to reduce demultiplexing latency, MED optimistically assumes the common case where the packet header has appropriate length and version, is contiguous in the first buffer, and has a correct checksum. Stages can invoke an early demultiplex verifier function, EDV, to verify MED's assumptions. EDV caches the veri-

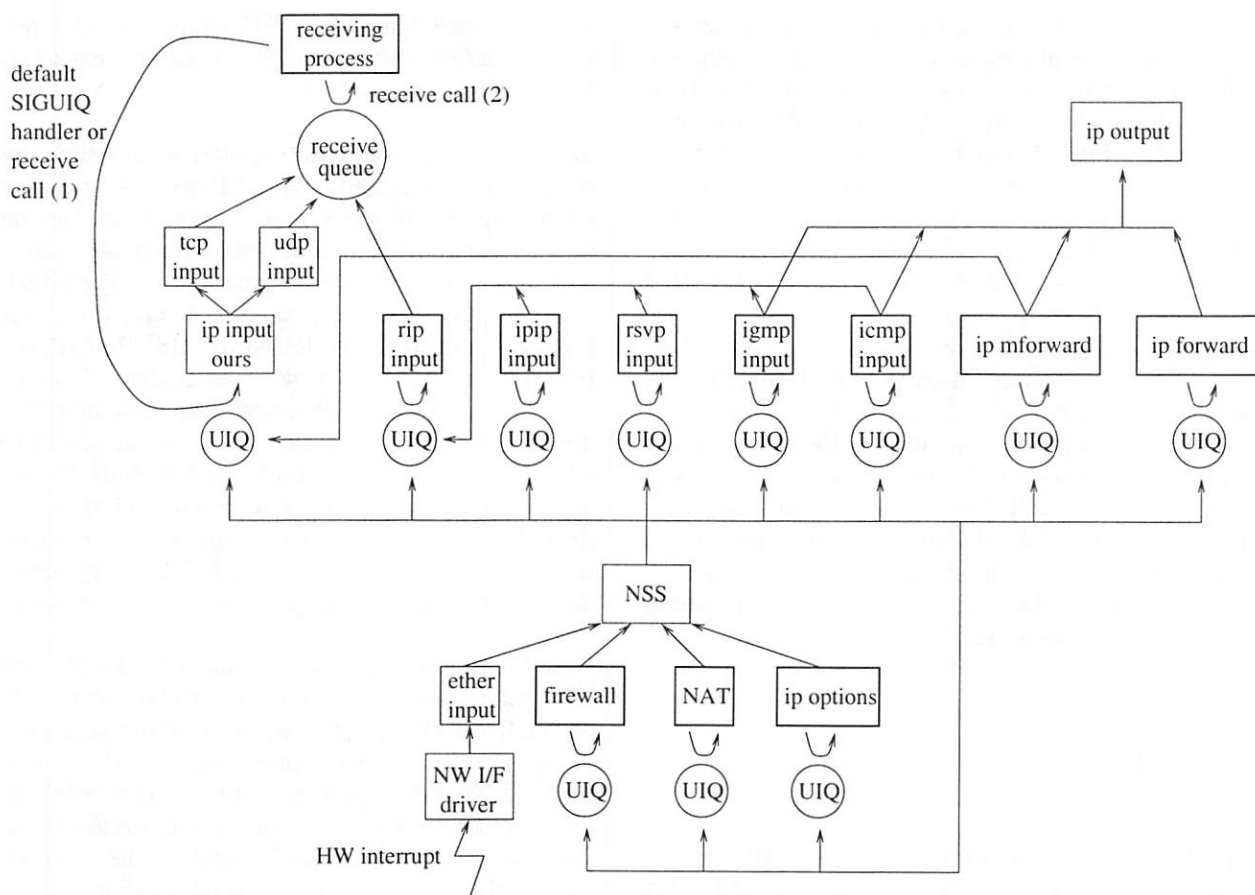


Figure 3: SRP processes received packets in the context of the receiving process, either in the default SIGUIQ signal handler or in receive calls. System processes with CPU reservations implement gateway protocol functionality.

fication in a flag in the packet's first buffer.

SRP buffers packets as follows. NSS invokes MED and determines the socket and UIQ pointed by the returned PCB. If the total amount of buffering used for the packet plus the socket's UIQ and receive queue exceeds the socket's receive buffer limit, NSS drops the packet; otherwise, NSS enqueues the packet in UIQ.<sup>3</sup> By demultiplexing and, in overload conditions, dropping packets *before* further processing them, SRP avoids receive livelock.

### 5.3 Packet notification and delivery

This subsection describes how protocols and applications receive notification and delivery of packets from SRP.

<sup>3</sup>Receive buffer limits can be set using the `setsockopt` system call.

When SRP's NSS demultiplexes and enqueues a packet in a certain socket's UIQ, if there are processes waiting for that UIQ to be non-empty, NSS wakes up those processes; otherwise, NSS signals SIGUIQ to the socket's owner processes.<sup>4</sup>

SIGUIQ is a new signal whose default action is to dequeue packets from UIQ of the process's sockets and submit each packet to the respective socket's input function. However, any process can catch, block, or ignore SIGUIQ signals and, for example, defer protocol processing of received packets until a subsequent receive call.

On receive calls, the receiving process first dequeues

<sup>4</sup>If all owner processes are sleeping non-interruptibly, signal delivery will be delayed until the first owner process is woken up. However, because processes only sleep non-interruptibly while waiting for a short-term (e.g., disk) event, the resulting SIGUIQ delay should not exceed other scheduling delays also present in a multitasking environment.

unprocessed packets from the socket's UIQ and submits those packets to the socket's input function. In the case of TCP or UDP sockets, the input function is a modified version of `ip_input`, which calls a modified version of `tcp_input` or `udp_input`, which finally enqueues the packet in the socket's receive queue. (The modifications in `ip_input`, `tcp_input`, and `udp_input` replace the original demultiplexing operations by cheaper verifications of MED's optimistic demultiplexing.) The receiving process then checks the socket's receive queue. If the queue is empty, the process sleeps; otherwise, the process dequeues the data and copies it out to application buffers.

## 5.4 Protocol scheduling options

This subsection discusses SRP's scheduling alternatives for protocol processing.

SRP's default SIGUIQ signal handler makes protocol processing asynchronous, accommodating those TCP applications for which synchronous protocol processing would be inappropriate.

Any application can, however, catch, block, or ignore SIGUIQ. To catch SIGUIQ, an application specifies one of the application's functions, *f*, to be called asynchronously by the system when a packet destined to the application arrives. *f* has the option of (1) issuing a receive call immediately, making protocol processing asynchronous with respect to the application's main control flow, or (2) deferring protocol processing to a synchronous receive call in the application's main control flow.

Another option is to block SIGUIQ. If an application blocks SIGUIQ, delivery of this signal is delayed until the application unblocks SIGUIQ. A final alternative is to ignore SIGUIQ. If an application ignores SIGUIQ, the system generates no signal when a packet destined to the application arrives. In such case, protocol processing is synchronous, occurring only when the application issues a receive call.

Applications may exploit the flexibility of catching, blocking, or ignoring SIGUIQ, for example: to control how much CPU time they spend on protocol processing; to prevent being interrupted while they are processing some critical event; or to perform protocol processing only immediately before they need the received data, which may improve memory lo-

cality.

If SIGUIQ signals are ignored or blocked, the number of context switches is the same as in the conventional BSD approach (no context switching when packets arrive), but memory locality may improve (packet data first accessed immediately before the application needs the data). LRP realizes the same benefits for UDP, but not for TCP.

If SIGUIQ signals are processed by the default handler or caught, the number of context switches may be higher than that of BSD, depending on the scheduling policy. However, the number of context switches will usually be less than one per packet. The default handler processes multiple packets in a process's UIQs when the process is scheduled. Additionally, if scheduling is time-sharing or real-time priority-based, a receiving process will preempt the currently running process only if the receiving process has higher priority (otherwise, context switching cannot happen). But if the receiving process has higher priority and is blocked on the socket, it would usually preempt the running process also on BSD. Similar observations apply to LRP's TCP handling. However, compared to SRP, LRP incurs additional context switching between the asynchronous TCP kernel threads or process and the receiving processes. Memory locality is similar for BSD, LRP's TCP, and SRP with default or caught SIGUIQ: The packet data is accessed asynchronously during protocol processing, possibly disturbing the cache.

## 5.5 Advantages

Because SRP checks buffering limits before protocol processing and processes protocols in the context of the receiving processes, SRP avoids receive livelock, charges protocol processing to the correct processes, and allows Eclipse/BSD to enforce and honor CPU guarantees.

SRP also solves LRP's problems mentioned in Section 4. Because SRP requires only a signaling facility, it can be easily ported to most existing systems, including those that support neither kernel threads nor Resource Containers. SRP's multi-stage demultiplexing function supports gateways, including NAT and other gateway functionality that may require packets to be demultiplexed multiple times. SRP's signals allow all applications, including those that use TCP, to opt for synchronous or

Protocol processing	Throughput (Mbps)		Utilization (%)	
	ave	std dev	ave	std dev
FreeBSD	69.35	0.70	36.3	0.6
SRP/default SIGUIQ	67.88	0.77	36.2	0.4
SRP/ignored SIGUIQ	68.30	1.00	36.4	0.6

Table 1: SRP does not significantly alter FreeBSD's TCP throughput and CPU utilization.

asynchronous protocol processing, thereby possibly improving performance. Finally, SRP's interaction with time-sharing, real-time, and flat or hierarchical proportional-share CPU schedulers is straightforward. SRP processes protocols always in the context of and under the control of the receiving process, regardless of how that process is scheduled. SRP thus avoids the difficult assignment of real-time priorities or proportional shares to separate TCP threads or system-wide TCP processes.

## 6 Experimental results

This section presents an experimental evaluation of SRP.

The first two experiments show that SRP does not significantly hurt FreeBSD's networking performance (throughput, CPU utilization, and latency). For these experiments, we connected two PCs S and R to the same Fast Ethernet hub (100 Mbps). Host S has a 300 MHz Pentium II CPU and 32 MB RAM and runs FreeBSD. Host R has a 266 MHz Pentium II CPU and 64 MB RAM and runs FreeBSD, Eclipse/BSD with default SIGUIQ, or Eclipse/BSD with ignored SIGUIQ. Both hosts use Intel EtherExpress PRO 100 Ethernet cards. In the first experiment, we ran on S a sender application that sends 10 MB data to a receiver application on R, using TCP with 64 KB send and receive socket windows and no delayed acknowledgments. A compute-bound background application also ran on R, but the hosts and network were otherwise idle. We measured the TCP throughput and R's CPU utilization (estimated by the background application's rate of progress) during the data transfer, repeating the experiment ten times. Table 1 presents the averages and standard deviations of our measurements.

In the second experiment, we ran on S and R applications that send to each other packets of increasing length, using UDP with 64 KB send and receive

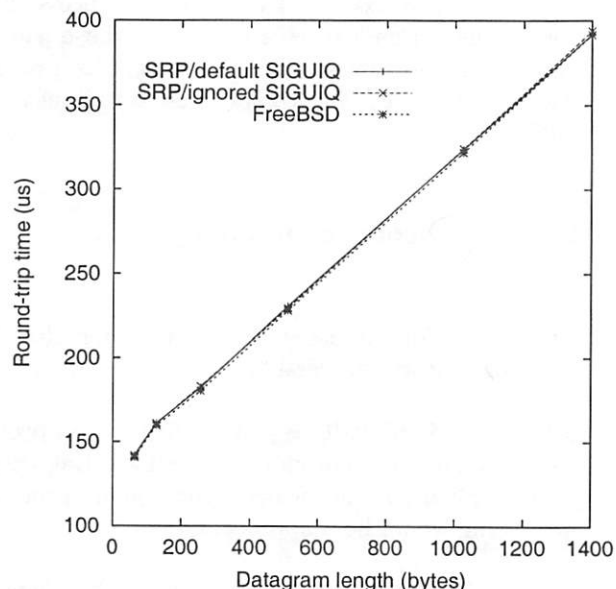


Figure 4: SRP does not significantly alter FreeBSD's UDP round-trip times.

socket windows. No other application ran on R. We measured the round-trip times and report on Figure 4 the averages of ten tries.

The results in Table 1 and Figure 4 suggest that SRP's performance penalties on FreeBSD are quite small or not statistically significant.

The third experiment demonstrates that SRP avoids receive livelock. In this experiment, host S is a Pentium Pro PC running FreeBSD, while host R is a PC running either FreeBSD or Eclipse/BSD on a 266 MHz Pentium Pro CPU with 64 MB RAM. The hosts were connected by Fast Ethernet at 100 Mbps. There was no other load on the hosts or network. A sender application on host S sent 10-byte UDP packets at a fixed rate to a receiver application on host R. When running on Eclipse/BSD, the receiver application used SRP's default SIGUIQ handler. We measured the application-level reception rate while varying the transmission rate, and report the av-



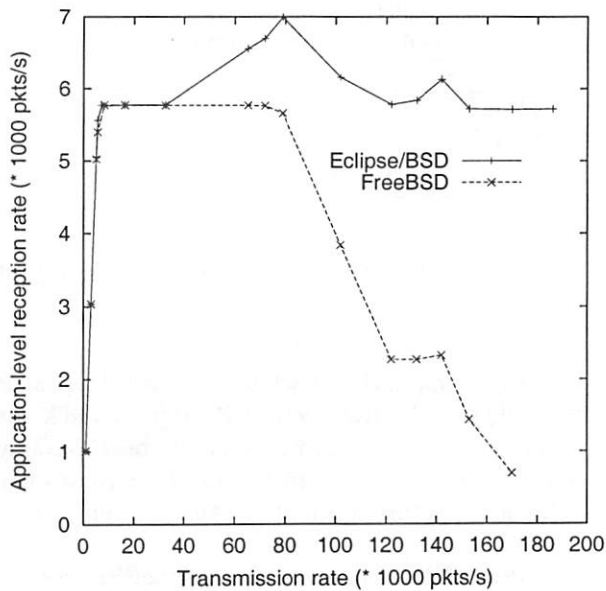


Figure 5: SRP prevents receive livelock in Eclipse/BSD.

erages of five runs. Figure 5 shows that, both on FreeBSD and on Eclipse/BSD, essentially all packets were received up to a transmission rate of about 5600 packets per second. Above a certain transmission rate, however, FreeBSD's reception rate collapses because of receive livelock. On the contrary, Eclipse/BSD's reception rate reaches a plateau and remains substantially at that level as the transmission rate increases. Eclipse/BSD's stability is due to SRP, which avoids receive livelock.

The fourth experiment shows that proportional-share scheduling is desirable in gateways that process application-specific code, in addition to forwarding packets. We used the *netperf* utility to measure TCP throughput between hosts A and B on two separate Fast Ethernet networks connected via a gateway G. Gateway G is a 266 MHz Pentium II PC with 64 MB RAM and running Eclipse/BSD, while host A is a 400 MHz Pentium II PC with 64 MB and running Linux, and host B is a 133 MHz Pentium PC with 32 MB RAM and running FreeBSD. In addition to IP forwarding, the gateway ran a variable number of instances of an application called *onoff*. After each time an *onoff* process runs for 11 ms, it sleeps for 5 ms. The IP forwarding process ran either with a 50% CPU reservation or with no reservation. The *onoff* processes ran with no CPU reservation. There was no other load on the hosts or network. Figure 6 demonstrates that, without a CPU reser-

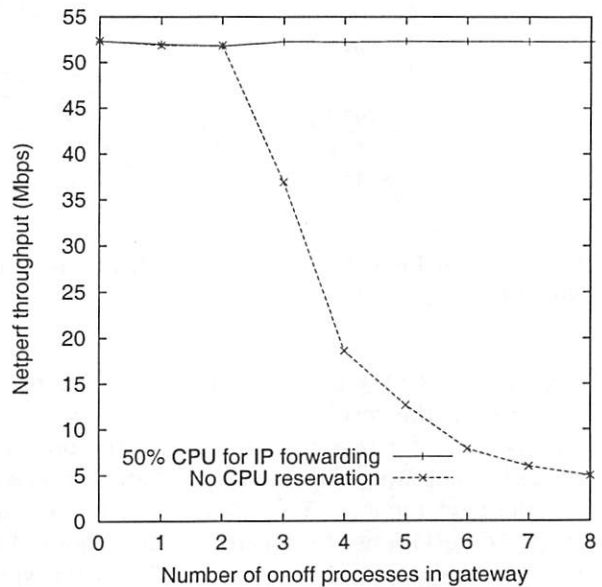


Figure 6: IP forwarding needs a CPU reservation to avoid losing performance due to other applications on the gateway.

vation for IP forwarding, other application load on the gateway can cause TCP throughput to collapse. On the contrary, an appropriate CPU reservation isolates IP forwarding performance from other load on the gateway.

The final experiment demonstrates that some applications can benefit from SRP's ability to defer protocol processing. In this experiment, a client application on host C continuously executed transactions each consisting of sending requests with 512 KB of random integer data to a server application on host S, and then receiving the server's reply of a few bytes. Host C is a 300 MHz Pentium II PC with 64 MB RAM and running FreeBSD, while host S is a 266 MHz Pentium II PC with 32 MB RAM and running Eclipse/BSD. The hosts were connected by a Fast Ethernet at 100 Mbps. Client and server applications communicated over a TCP connection between sockets with 512 KB send and receive buffers. Hosts and network were otherwise unloaded. The server application processed requests using one of three algorithms: compute five averages of the request data; view the request data as two sets of four 64 KB vectors and compute their 16 internal products; or select the *n*th largest number among the request data (using *partition* [13]). While processing these algorithms, the server application either used the default SIGUIQ handler or ignored the SIGUIQ

Application	Elapsed time (ms) with SIGUIQ				Improvement %
	default		ignored		
	ave	std dev	ave	std dev	
Averages	63.1	0.8	54.8	0.2	13.1
Internal products	50.7	0.2	46.5	0.4	8.3
Select nth	61.0	0.3	55.7	0.2	8.6

Table 2: On Eclipse/BSD, some applications can improve performance by catching, blocking, or ignoring SIGUIQ signals.

signal. We used the CPU's internal cycle counter to measure, in the server application, the time interval necessary for sending the reply to the previous request, computing the current request, and receiving the next request. We report the averages and standard deviations of ten runs. Table 2 shows that this server application runs up to 13% faster when it ignores SIGUIQ, making protocol processing synchronous. This improvement is due to better memory locality when protocol processing is performed only immediately before the data is needed.

## 7 Related and future work

We are not aware of previous reports about experiences in porting LRP or Resource Containers to other systems, especially systems that offer QoS guarantees via proportional-share scheduling, such as Eclipse/BSD.

This paper describes only how Eclipse/BSD processes packets received from a network. Other papers describe Eclipse/BSD's overall architecture and application programming interface (`/reserv` [7]), CPU scheduler (MTR-LS [5]), disk scheduler (YFQ [6]), and network output link scheduler (Bennet and Zhang's WF<sup>2</sup>Q [2, 3]). Eclipse/BSD is easy to use: Even unmodified legacy Unix applications can automatically run with appropriate QoS guarantees under Eclipse/BSD [8].

Nemesis [18] is an operating system built from scratch according to a radical new architecture designed to prevent *QoS cross-talk*, that is, one application's interference in another application's performance. Most Nemesis services, including TCP/IP, are implemented as libraries that are linked with applications. Therefore, services are performed in the context of and charged to the respective ap-

plications, similarly to what is achieved by SRP's SIGUIQ signals. However, SRP's signals and kernel-mode signal handler may be easier to port to today's mainstream systems, which typically have a monolithic architecture quite unlike that of Nemesis.

Because LRP is not available on FreeBSD, we were unable to compare SRP and LRP directly. Such comparison would be interesting future work.

## 8 Conclusions

We proposed a new mechanism, SRP, whereby packet arrival sends a signal to the receiving process. The default handler of this signal performs protocol processing on the packet, but the receiving process may catch, block, or ignore the signal and defer protocol processing until a subsequent receive call. In any case, protocol processing occurs in the context of the receiving process and is correctly charged. Our experiments show that, like LRP, SRP avoids BSD's receive livelock. However, SRP has the advantages of being easily portable to systems that support neither kernel threads nor Resource Containers, such as FreeBSD; giving applications control over protocol scheduling; using a multi-stage demultiplexing strategy that supports gateway functionality; and easily enabling real-time or proportional-share scheduling.

## Acknowledgments

We thank John Bruno and Banu Özden for valuable discussions during the design phase of this work. We also thank the anonymous referees and our paper's shepherd, Liviu Iftode, for their useful comments.

## References

- [1] G. Banga, P. Druschel and J. Mogul. "Resource containers: A new facility for resource management in server systems," in *Proc. OSDI'99*, USENIX, Feb. 1999.
- [2] J. Bennet and H. Zhang. "WF<sup>2</sup>Q: Worst-Case Fair Weighted Fair Queueing," in *Proc. INFOCOM'96*, IEEE, Mar. 1996, pp. 120-128.
- [3] J. Bennet and H. Zhang. "Hierarchical Packet Fair Queueing Algorithms," in *Proc. SIGCOMM'96*, ACM, Aug. 1996.
- [4] R. Braden, L. Zhang, S. Berson, S. Herzog and S. Jamin. "Resource ReSerVation Protocol (RSVP) - Version 1 Functional Specification." IETF, RFC 2205, Sept. 1997.
- [5] J. Bruno, E. Gabber, B. Özden and A. Silberschatz. "The Eclipse Operating System: Providing Quality of Service via Reservation Domains," in *Proc. Annual Tech. Conf.*, USENIX, June 1998, pp. 235-246.
- [6] J. Bruno, J. Brustoloni, E. Gabber, B. Özden and A. Silberschatz. "Disk Scheduling with Quality of Service Guarantees," in *Proc. ICMCS'99*, IEEE, June 1999, vol. II.
- [7] J. Bruno, J. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. "Retrofitting Quality of Service into a Time-Sharing Operating System," in *Proc. Annual Tech. Conf.*, USENIX, June 1999, pp. 15-26.
- [8] J. Brustoloni, E. Gabber, A. Silberschatz, and A. Singh. "Quality of Service Support for Legacy Applications," in *Proc. NOSS-DAV'99*, June 1999, pp. 3-11. Available at <http://www.nossdav.org/>.
- [9] K. Calvert, S. Bhattacharjee, E. Zegura and J. Sterbenz. "Directions in Active Networks," in *Communications Magazine*, IEEE, 1998.
- [10] J. Case, M. Fedor, M. Schoffstall and J. Davin. "A Simple Network Management Protocol (SNMP)," IETF, RFC 1157, May 1990.
- [11] K. Cho. "A Framework for Alternate Queueing: Towards Traffic Management by PC-UNIX Based Routers," in *Proc. Annual Tech. Conf.*, USENIX, June 1998.
- [12] Cisco. "FlowCollector Overview," at [http://www.cisco.com/univercd/cc/td/doc/product/rtrmgmt/nfc/nfc\\_3\\_0/nfc\\_ug/nfccover.htm](http://www.cisco.com/univercd/cc/td/doc/product/rtrmgmt/nfc/nfc_3_0/nfc_ug/nfccover.htm)
- [13] T. Cormen, C. Leiserson and R. Rivest. "Introduction to Algorithms." MIT Press, Cambridge, MA, 1990.
- [14] P. Druschel and G. Banga. "Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems," in *Proc. OSDI'96*, USENIX, Oct. 1996, pp. 261-275.
- [15] K. Egevang and P. Francis. "The IP Network Address Translator (NAT)," IETF, RFC 1631, May 1994.
- [16] Flux Research Group. "The OSKit," at <http://www.cs.utah.edu/projects/flux/oskit/>.
- [17] P. Goyal, X. Guo and H. Vin. "A Hierarchical CPU Scheduler for Multimedia Operating Systems," in *Proc. OSDI'96*, USENIX, Oct. 1996, pp. 107-121.
- [18] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns and E. Hyden. "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications," in *JSAC*, 14(7), IEEE, Sept. 1996, pp. 1280-1297.
- [19] M. McKusick, K. Bostic, M. Karels and J. Quarterman. "The Design and Implementation of the 4.4 BSD Operating System," Addison-Wesley Pub. Co., Reading, MA, 1996.
- [20] J. Mogul and K. K. Ramakrishnan. "Eliminating Receive Livelock in an Interrupt-Driven Kernel," in *Proc. Annual Tech. Conf.*, USENIX, 1996, pp. 99-111.
- [21] L. Peterson, S. Karlin and K. Li. "OS Support for General-Purpose Routers," in *Proc. HotOS'99*, IEEE, Mar. 1999.
- [22] P. Srisuresh and D. Gan. "Load Sharing using IP Network Address Translation (LSNAT)." IETF, RFC 2391, Aug. 1998.
- [23] W. R. Stevens. "TCP/IP Illustrated," vol. 1, Addison-Wesley Pub. Co., Reading, MA, 1994.





# DITools: Application-level Support for Dynamic Extension and Flexible Composition\*

Albert Serra, Nacho Navarro, Toni Cortes  
*Departament d'Arquitectura de Computadors*  
*Universitat Politècnica de Catalunya*  
{alberts,nacho,toni}@ac.upc.es

## Abstract

Today, operating systems set-up process images from executable files using fixed rules. Programs are restricted to run in essentially the same environment at every execution. However, we believe that this behavior is not always convenient, and that many times it is interesting to make variations to the execution environment, either to introduce new functionality or to specialize critical services, even when their source code is not available. This problem can be mitigated through application-level extensibility and flexible composition of binary modules.

In this paper, we describe DITools an application-level tool that supports dynamic interposition on dynamically-linked procedure-call boundaries. This tool enables both global and per-module dynamic interposition. We also present a detailed use of DITools and various short examples of extensions.

## 1 Introduction

Modifying programs, libraries and operating systems becomes a difficult problem when the source code is not available, and this is a common situation. Vendors are quite reticent to give away the source code of their applications and/or operating systems. On the other hand, there are many situations in which these modifications would be very useful. Let us examine some examples.

Tracing and monitoring applications is a common technique used to learn the behavior of applications. To perform this task, some code has to be added to the program to trace or monitor the desired events. If the source code is not available, there is no easy way to do it.

Another problem arises when trying to enhance proprietary applications to take advantage of parallel resources such as multiple CPUs. Let us suppose an application that invokes many time-consuming

operations from a sequential mathematical library, for example large FFTs. If we do not have the source code, changing the implementation of the FFT to make it a parallel one, would probably mean to write a whole new library. If we only want to enhance the FFT, it is going to be very difficult without the source code of the application or the library.

If we focus our attention on the operating system execution environment, it is very difficult to modify its functionality (for instance, to build a new file system) without OS support. Even if there is some OS support, we will probably need the help of the system administrator to install our enhancement. There is no easy way to do a rapid prototyping that is being used by many applications without the help of the system administrator.

Finally, sometimes we would like to modify an application to send and receive the data in an encrypted form. If we do not have the source code of the application or library, this modification will be a big problem.

In this paper, we present a tool that can be used to solve all the above problems. This tool provides transparent user-level extensibility and flex-

---

\*This work has been supported by the Ministry of Education of Spain (CICYT) under contract TIC98-0511, and by the Comissionat per a Universitats i Recerca de la Generalitat de Catalunya under the grant FI96-3088

ible composition within applications. This allows any user to load new code and to interpose it between the call and the definition of functions, making possible to modify or extend programs without rebuilding them. We also describe several examples in which we use the tool to solve the aforementioned problems.

This paper is structured as follows. Section 2 gives an overview of our approach to dynamic extension and flexible composition, describing the framework and the tool. Section 3 discusses performance issues and presents an evaluation. Section 4 contains the examples. Section 5 reviews related work and, finally, Section 6 summarizes and concludes the paper.

## 2 Dynamic Extension and Flexible Composition

In order to build the process image for a given executable file, modern operating systems need to glue together multiple modules (dynamically-linked libraries) at program load time. This happens because programs are not self-contained in terms of functionality.

The system provides service definitions to complete the image. These definitions are located using fixed resolution policies. This results in essentially the same execution environment at every run of the same program. However, as we illustrate in the introduction, there are situations in which it can be very helpful to make variations in this image. Moreover, these variations should not be limited by source code availability, nor by arbitrary rules embedded in operating system components. This motivates our interest on making the process of building the runnable image more controllable by users and programs themselves.

The goal of our research is to enable ‘ad-hoc’ execution environments, by allowing applications to build appropriate execution environments for themselves, according to performance requirements, efficiency concerns and functionality needs. This covers the improvement or tuning of services (e.g. service specialization or result ‘memoization’ – see [17]), the addition of new functionality to do data stream processing (e.g. encryption or compression) or to perform new tasks (e.g. cooperation with resource

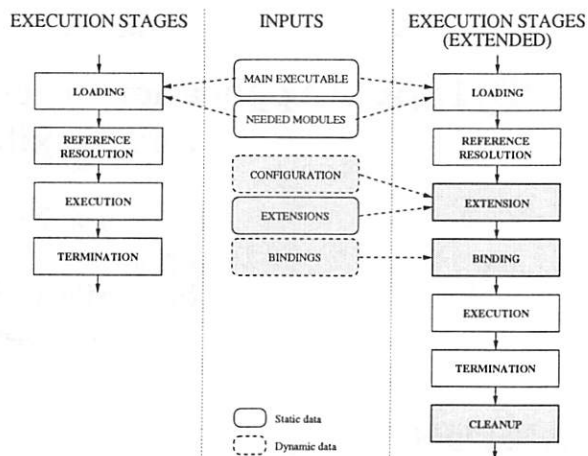


Figure 1: Usual and extended execution stages

managers), and even restructuring the execution environment (e.g. code co-location and distribution).

In Figure 1 we show the phases in which we conceptually divide the process of executing a program. Two scenarios are shown: the usual one (left side) and the extended one (right side). The meaning of each stage is as follows: *loading* brings all the required modules into the address space, *reference resolution* finds definitions for unresolved references, *extension* introduces new modules in the address space, *binding* allows to customize bindings, *execution* enters application code, *termination* exits application code and *cleanup* shuts down extensions. For each stage, we show which inputs determine its behavior. The stages shaded and surrounded by thick lines are provided by our tools.

In the *extension* stage, we check for the need of extending the image using configuration information that can change at every run. If extensions are needed, the right modules are introduced in the process image. Then, the *binding* stage arranges these modules to be executed by setting up bindings appropriately. This stage is independent from the previous one. Therefore, it is possible to modify bindings without loading extensions.

The tools described in this paper introduce an extension to the runtime loader and linker. This extension provides basic functionality to load and bind unforeseen modules and services, as well as to re-

consider actual bindings. This functionality works entirely at user-level and complements the traditional dynamic loading and linking services. This is achieved by means of additional stages after program loading, devoted to adapt/extend the execution environment. These tools also export services, making them available during execution, to allow on-the-fly adaptation.

These tools are being used by various projects in our department for trace collection, scheduling research and I/O research. They have helped us to test the stability and validity of our services and abstractions.

## 2.1 Environment

Modern operating systems promote the use of shared libraries for efficiency. Linking against shared libraries results in smaller program sizes on disk, and also requires less physical memory at run time. Moreover, shared libraries can be fixed and improved without rebuilding executables.

The use of shared libraries leads to executable files containing unresolved references. At program loading time, the system should load the libraries upon which executable files depend, and then it should fix unresolved references to point to the right definitions. This is called *dynamic linking*.

Dynamic linking requires cooperation between development tools, object file formats and operating system components. For example, a common technique for supporting dynamic linking is by making indirections through *linkage tables*, which are placed in the executable file by the static linker, and then used by the dynamic linker at load time. Usually, the system's Application Binary Interface (ABI) defines the object file format and these data structures.

During the last years, many vendors are adopting the System V r4 ABI [4], or at least parts of it. IRIX, Solaris, Linux, FreeBSD or HP-UX currently support ELF as the object file format and use similar dynamic linking conventions. ELF is the file format originally introduced in System V r4.

The standardization of binary interfaces makes possible to take a generic approach for extending the available functionality (we currently have a running

version for IRIX and Linux sharing a 70% of source code), as well as for manipulating bindings both at program loading time or at run time.

## 2.2 Exploiting dynamic loading

Shared libraries make dynamic loading of code easier. In fact, many operating systems provide interfaces to load this kind of binary modules at run time. We exploit these services to introduce extension code in the process image (although we improve some aspects of conventional dynamic loading, as explained in following sections).

The mechanisms used to gain control at program startup can vary from system to system, but they are reasonably uniform and widespread in the UNIX world. In most cases, this is as simple as setting an environment variable (`LD_PRELOAD` in Linux and Solaris or `_RLD_LIST` in IRIX, for example) pointing to the module to be loaded, and to declare an 'init routine' in this module. At program startup, this module will be loaded within the process image, and the init function will be executed before the program itself. For security reasons, the system loader disables this feature when loading `setuid` programs.

## 2.3 Dynamic interposition

Conceptually, 'interposition' is the addition of functionality at the midst of an existing interface boundary. This mechanism is appropriate for binding extensions, because it exploits existing interface boundaries to attach new services while preserving old ones.

We deal with interposition at procedure-call boundaries. That is, we add functionality in the program execution path, between references to procedures and the procedures themselves.

To achieve this kind of interposition dynamically, at run time, we need to detect references to procedures and to be able to change their target definitions within the process image. Dynamic linking draws a clean boundary between external references and their definitions, by means of the linkage tables and other data structures described above. In this work we exploit these data structures for interposition purposes.

## 2.4 DITools

The infrastructure required to manage dynamic loading of extensions and interposition is provided by DITools (our Dynamic Interposition Tools). DITools allows modifying the execution environment of dynamically-linked executables at every run, either to select among different service implementations, as well as to accommodate unforeseen functionality. Using this infrastructure, programmers are allowed to change bindings, redefine symbols, load new modules or remap global variables.

### Architecture

DITools is structured around a runtime module (DI runtime) that cooperates with the dynamic loader and linker to support extension and flexible composition. Once the program, together with all its dependencies, has been loaded by the system, the DI runtime gains control and performs some post-load processing (see figure 1), according to the needs for this individual run. This is done using existing mechanisms, as mentioned in section 2.2.

It is interesting to note that this tool works entirely at user level, without kernel support, and without administrator privileges. The scope of the tool can be easily controlled, so it can manage from single programs to entire sessions and multiple users.

DITools processing can be driven by a configuration file or it can be driven using a programming interface, or some combination of both approaches. DITools allows two main classes of operations: loading of new functionality and manipulation of bindings. Both kinds of operations will be explained in detail in the following sections.

### Loading of new functionality

During the extension stage, as well as at run time, DITools can load arbitrary extension modules (also called backends in our framework) within the process image. These modules can be declared in a configuration file, to keep the executable file isolated from this kind of 'volatile' decisions.

We have adopted the shared-library model for these extensions, which means that our extensions are

shared libraries themselves. This has two basic advantages. The first one is that we can use the existing development tools. And second, as our extensions are also shared libraries, we can use DITools recursively to extend them.

The support provided by DITools allows modules to be loaded multiple times within the same process (i.e. replicated) without symbol collisions.

### Rebinding and redefinition

We exploit the dynamic-linking data structures for interposition purposes. DITools supports explicit manipulation of bindings between dynamically-linked references and definitions exported by modules. It currently implements two mechanisms: the first one allows changing the target for a given reference (we call it *rebinding*), and the second one allows 'wrapping' (i.e. mediate all the uses of) a given definition (we call it *redefinition*). While the effects of the first mechanism are selective and leave the original definition untouched, the second one affects all the uses of a definition, effectively hiding it to the outside. Using rebinding we can achieve *selective overriding* of definitions, while using redefinition we are doing *global overriding* of definitions.

References and definitions are identified by a pair (module, name). So, a single rebinding can be specified as follows:

```
(program,read) -> (mymodule,myread)
```

Meaning that references to 'read' done by the module 'program' should point to the definition 'myread' in the module 'mymodule'.

As can be observed, this mechanism is independent of conventional resolution policies based on name matching. This makes possible the coexistence of definitions with the same name in different loaded modules.

Acting over references and definitions individually also proves to have its advantages. On the one hand, we can avoid affecting all the uses of a definition when inserting new functionality. On the other hand, we do not require to subclass the entire library in order to interpose on its interface.



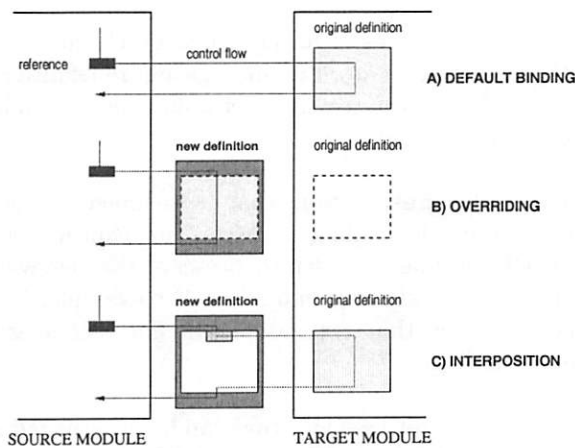


Figure 2: Scenarios after rebinding references

## Interposition on top of DITools

Figure 2 shows some possible scenarios after rebinding. The upper one (A) depicts how bindings are by default, in which pointers in the reference side make possible to reach definitions residing in another module. The central one (B) shows how changing bindings in the reference side can be used to override definitions in a per-module basis. The lower one (C) shows how to achieve interposition, by changing the binding while providing another one to invoke the previous definition.

Therefore, DITools can be used for interposition of extensions. This requires changing a binding on the reference side to point to the extension, and another one within the extension code to point to the original definition. For example:

```
(program,read) -> (mymodule,myread)
(mymodule,read) -> (libc,read)
```

The above lines add 'mymodule.myread' as the target for references to 'read' being done by the module 'program'. References to 'read' coming from 'mymodule' are conveyed to 'libc.read'. Although this last rebinding is unnecessary in the case in which it refers to the conventional definition, we show it for clarity.

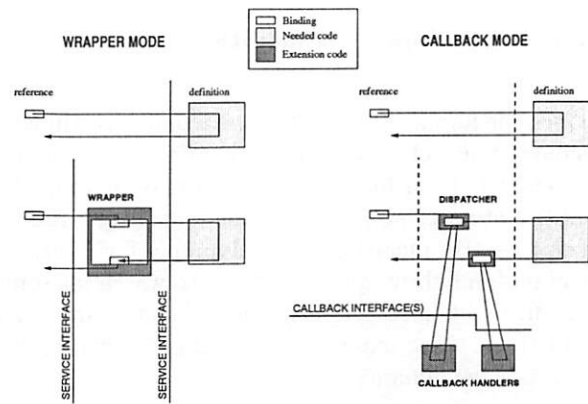


Figure 3: Wrapper and callback modes of extension execution

## Extension execution modes

To simplify design and implementation of extensions, DITools allows two modes of extension execution: *wrapper* and *callback*. The *wrapper* mode can be seen as 'plain interposition'. This mode requires providing routines that match the interface on which they will be interposed. The *callback* mode is a refinement that allows a routine to be invoked regardless of the interface being intercepted. This refinement is especially convenient when building extensions for monitoring purposes, as well as when the interface is not known.

Both modes are depicted in Figure 3. The *wrapper* mode is shown in the left side of this figure. Observe that the wrapper is invoked by the application as the effective service definition. Therefore, the wrapper should present the same interface to the application. The *callback* mode appears in the right side of the figure. In this mode, DITools transparently interposes a *callback dispatcher* instead of the extension routine. This dispatcher will invoke the routines 'aside' of the service definition, without need of knowing the implied interface. Callback handlers have a fixed interface (as illustrated in Section 4), which is determined by the callback dispatcher that calls them. The dispatcher is also the responsible of invoking the original service definition, as if nothing happened.

Extensions can provide their own dispatchers to enable different ways of invoking extension code, for instance to allow multiple handlers for the same event, as in other frameworks [14] [15].

## Dynamic adaptation facilities

Once the application has been started, it is still possible to make changes in the execution environment, given that both loading and bind manipulation facilities are available through the interface of the DI runtime. By means of these dynamic facilities, we can design lightweight extensions to watch for some conditions before activating new functionality. At run time, these extensions can also deactivate the new service dynamically.

Let us suppose that we have installed an optimized read path for files with read-only access. The following code illustrates how to deactivate the optimization when conditions change. The wrapper has been installed to be activated in response to references to 'open()'. It checks for the O\_RDONLY flag, and deinstalls the new paths when a file is opened with flags that can compromise the extension:

```
int rdopen_wrapper(char *file, int flags)
{
    if (flags & O_RDONLY)
        di_rebind("program", "read",
                  "extension", "rdo_read");
    else
        di_rebind("program", "read",
                  "extension", "rdwr_read");

    return(open(file, flags));
}
```

The DITools interface provides rebinding, symbol redefinition, interrogation about current bindings and loaded modules, among other operations. This interface is described in more detail in a research report [18].

## Process management facilities

**Startup:** When extension modules are loaded, DITools allows them to initialize before being invoked by the process. This can be used to allocate data structures or to set-up some bindings dynamically.

**Cleanup:** At the end of execution (for instance, when the process exits or execs another binary), DITools gives a chance for doing cleanup work within extension modules (e.g. write data structures to disk, close any private handle or free other

allocated resources). During extension cleanup, the DI runtime shuts down rebindings and redefinitions in order to guarantee that extensions do not indirectly trigger themselves.

**Image changes:** Actions that cause image changes (e.g. dynamic-loading requests) are captured by the DI runtime in order to preserve the behavior expected by extension modules. For example, DITools checks that new modules do not affect existing rebindings.

**Fork:** A default behavior that can be deactivated if required is that forked processes inherit the DI runtime, as well as rebindings and redefinitions existing in the forking process.

## 2.5 Considerations

The support described above has some natural limitations. First of all, the mechanism cannot be used in statically-linked binaries. On the other hand, the set of available dynamically-linked procedure calls may not be enough for some uses that require customizing pieces smaller than functions. Also, some bindings may not be available for manipulation due to static optimizations. In these cases, the scope of the tools can be complemented using binary rewriting techniques (e.g. Parady's dynamic instrumentation [22]) to invoke the backend for events other than dynamically-linked procedure boundaries. However, some systems also use linkage tables to support Position Independent Code. This makes possible to use DITools to control intra-module bindings in addition to external references.

Finally, there are also not-so-obvious considerations to take into account when using all the facilities provided by DITools. The design of existing library services may be inappropriate for dynamic rebinding, because they may keep state between invocations. On the other hand, there are situations in which modules can shortcut linkage tables used in cross-module invocations (e.g. by assigning an address to a pointer and then invoking it directly).

## 3 Performance

The DITools infrastructure enables qualitative benefits, namely richer functionality and ease of ex-

tension, that are hard to measure. In this section, we will focus on the overhead of adding new functionality, regardless of the beneficial counterparts, and we found it quite reasonable. Nevertheless, we believe that performance is not always the primary concern.

We have designed a worst-case experiment (the ‘Forward’ experiment) that redefines all the available dynamically-linked definitions within a given process to a simple extension that merely forwards the call to the original definition. This includes all definitions of the standard C run-time library as well as those included in any other library used by the program. By comparing the results to the execution of the unmodified program (the ‘Baseline’ experiment), we evaluate the overhead of using DITools. Typical uses of DITools (like those described in section 4) only need to interpose code to very few calls.

The programs used in our tests come from the SPEC95 benchmark suite, using the ‘train’ input dataset. Our experimental environment is based on a 64-processor Origin2000 system, from Silicon Graphics Inc. This machine runs the IRIX Operating System, release 6.5.3.

In programs enhanced by our infrastructure we expect to observe two effects on performance: an increase in startup time due to extra processing, and some overhead during execution due to the additional indirection.

Results summarized in Table 1 come from the average of 4 executions of each benchmark, running on a dedicated processor of the machine, to minimize the effects of cold start and interferences from other processes. This table shows, for each program, the number of statically available dynamically-linked references (*static hooks*), how many times these hooks are triggered at runtime (*dynamic activation count*), the elapsed time for the unmodified execution environment (*baseline*), and the elapsed time when invoking the empty wrapper at every call.

The *static hooks count* column gives an idea of the work done by the DI runtime at startup. As many bindings as listed in this column are modified to invoke the extension. This classifies the benchmarks in three groups, according to the number of exposed hooks. Benchmarks that need the same libraries expose the same number of hooks. Given that the *startup* time is basically constant (around 30 mi-

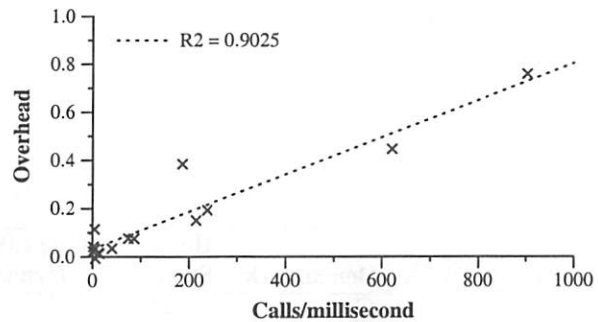


Figure 4: Overhead vs. calls per millisecond

croseconds/hook), we do not show it in the table. It ranges from 140 ms for the first five benchmarks (3,500 exposed hooks) to 150 ms for the last 8 benchmarks (5,600 exposed hooks).

Last three columns show the elapsed time for *baseline* and *forward*, and the corresponding *overhead*. The overhead of DITools is proportional to the dynamic count of hook calls. In many cases it is less than 10%. However, benchmarks that are making a high number of calls to the extension, relative to their execution time, become more affected by interposition. This is simply the scaled effect of this high number of calls per millisecond (c/ms). For instance, if we compute this number for perl and su2cor, we obtain 902 c/ms for perl and 621 c/ms for su2cor. Both programs follow the correlation between calls per millisecond and overhead seen in the other programs. Take, for instance, turb3d, which has an overhead of 3%. It does 40 c/ms, so this makes 1,013 c/ms for 76% and 613 for 46%, which are close to the values observed for perl and su2cor. The measured correlation coefficient between overhead and calls per millisecond is 0.9 (see the regression plot in Figure 4). It is worth to mention that we are describing a worst-case experiment, in which the extension interposes to all the available hooks.

Table 2 depicts the space requirements of the DI runtime and the other extensions used in this evaluation, as well as the average size of the benchmarks. This table shows, for each module, its static size in disk as well as the size of all the virtual memory regions required to hold its code and data within the process address space at run-time. The ‘count’ extension has been used to compute the columns labeled ‘hook counts’ in Table 1, the ‘forward’ extension corresponds to the ‘forward’ experiment, and the ‘monitoring’ extension to the ‘monitoring’ experiment.

Benchmark	Hook counts (/1,000)		Elapsed time (ms)		Overhead %
	Static	Dynamic	Baseline	Forward	
<i>Go</i>	3.5	5.8	5,002.3	5,115.0	2%
<i>M88ksim</i>	3.5	4.8	864.6	858.8	0%
<i>Gcc</i>	3.5	289.3	1,555.9	2,154.7	38%
<i>Compress</i>	3.5	0.7	181.5	202.2	11%
<i>Ijpeg</i>	3.5	12.3	2,385.6	2,486.7	4%
<i>Li</i>	3.9	4.3	975.8	991.6	2%
<i>Perl</i>	3.9	12,335.0	13,662.2	24,051.4	75%
<i>Tomcatv</i>	5.6	3,412.1	39,599.9	42,582.3	8%
<i>Swim</i>	5.6	5,284.1	2,466.2	2,836.9	15%
<i>Su2cor</i>	5.6	24,504.6	39,447.5	57,090.1	45%
<i>Mgrid</i>	5.6	319.0	23,107.2	23,425.7	1%
<i>Applu</i>	5.6	4.3	1,020.8	1,053.8	3%
<i>Turb3d</i>	5.6	1,600.7	39,811.9	41,159.9	3%
<i>Fpppp</i>	5.6	50.1	676.9	728.2	8%
<i>Wave5</i>	5.6	2,002.2	8,417.1	10,039.7	19%

Table 1: Impact of interposition on program execution

Module	Static size	Dynamic size
<i>DI runtime</i>	64K	1.5M
<i>Count extension</i>	29K	0.5M
<i>Forward extension</i>	30K	0.5M
<i>Monitoring extension</i>	30K	1M
<i>SPEC average</i>	430K	10M

Table 2: Space requirements of modules used in this evaluation



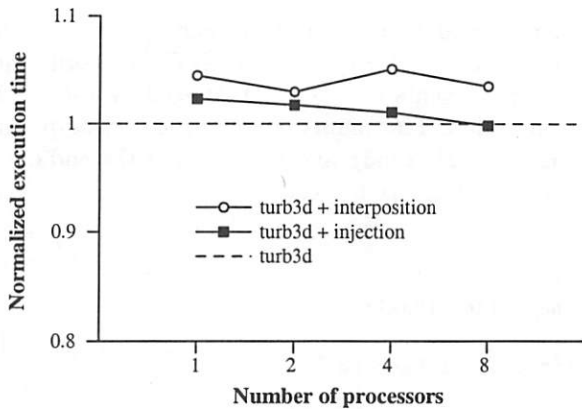


Figure 5: Impact of a fully-featured extension on execution time

The dynamic sizes have been obtained by measuring the number of pages allocated in the virtual process address-space, and then using the page size (16K) to compute the dynamic size. The space overhead at run-time is about 15%. We should take into account that benchmarks are running with a relatively small input ('train').

The second experiment (the 'Monitoring' experiment) compares the performance of a fully-featured extension using DITools, against the same functionality introduced by changing the source code. The experiment introduces a performance monitoring extension that collects the execution trace of a parallel program. This trace contains thread creation, thread joining and synchronization events. More information on this experience can be found in another paper [10].

For this experiment, we use a parallelized version of the turb3d benchmark. Figure 5 shows the impact of both alternatives. The dashed line represents the normalized execution time for the uninstrumented version of this benchmark, using from 1 to 8 processors. Solid lines represent the normalized execution time for the instrumented versions. As can be observed, interposition-based instrumentation performs comparably to the static code modification approach. In both cases, the overhead is less than 5% of the execution time for any number of processors.

At a first glance, one can think that all these enhancements always come at the price of performance. In another experiment, we used interposition to enhance the performance by caching the

results of frequently used functions [17]. In this way we were able to reduce the execution time by 8% for those library functions. This demonstrates that extending the execution environment not always degrades performance.

Finally, remember that, although this section focuses on the costs of adding new functionality using DITools, this infrastructure is intended to support extension and flexible composition. Therefore, performance will not always be the primary concern. In many cases, the infrastructure will not be used to *add* functionality but to *select* among multiple implementations (thus, not adding extra indirections), and it may even pay using it in terms of performance.

## 4 Examples

In this section, we discuss how to build a couple of useful modules using DITools to facilitate a better understanding of the framework. We begin with a full example, and then we give short ideas about how to use DITools to build some additional extensions.

### 4.1 Program monitoring

Interposition can be used to understand the behavior of programs by capturing information of service invocations. In this example, we describe an extension used to monitor interactions between a program and its runtime. This extension will provide an execution profile containing function entry and exit information, and the corresponding timestamps. Once processed, the output of the extension can give a profile like the following one, coming from the execution of the NAS BT benchmark on a SGI Origin 2000:

```
...
2623262 +178 x_solve_
2623486 +175 lhsx_
2638509 -175
2638738 +186 x_solve_cell_
2673173 -186
2673412 +185 x_backsubstitute_
2674186 -185
2674361 -178
2674493 +191 y_solve_
```

```

2674630      +176 lhsy_
2678248      -176
2678449      +199 y_solve_cell_
2741176      -199
2741403      +198 y_backsubstitute_
2741949      -198
2742171     -191
...

```

The first column in this profile contains the event timestamps in nanoseconds, followed by a sign (+/-) indicating procedure entry/exit, then the event identifier, and finally the event name (in this case, a procedure name). Tabulation represents invocation nesting.

In this extension, we decided to use the callback mode (see 'Extension execution modes' in section 2.4), because it allows to capture all the references using only two handlers.

The steps to be followed to build and install this extension can be summarized as follows:

**1-Write the event selector function.** When using the DITools callback dispatcher, the user should provide a routine (`di_callback_required`) in the backend to select which hooks will trigger the user-provided callback handlers, and to give an identifier for each possible event. The function will be invoked at extension time (see section 2 for a description of the stages) once for each potential event. A return value of zero can be used to ignore the event. The event identifiers will be passed at execution time to the callback handler.

```

int di_callback_required(char *func)
{
    static int event_id=0;

    event_id++;
    funcs[event_id]=RECORD_FUNC_NAME(func);
    return event_id;
}

```

**2-Implement callback handlers and support routines,** i.e. the code which should record the time and the event. Callback handlers will be invoked by the callback dispatcher before (`di_pre_handler`) and after (`di_post_handler`) dynamically-linked hooks selected by the event selector function. Callback handlers receive two arguments from the dispatcher: a virtual processor

identifier and an event identifier. In this example, we use a macro (`PUT_EVENT`) to record time-stamped events in a buffer managed by the extension code. This buffer is set-up at startup time (`di_init_backend`), and processed at the end of execution (`di_fini_backend`).

```

event_t *buffer;
char *funcs[MAX];

int di_init_backend()
{
    buffer=ALLOCATE_BUFFERS();
    return buffer!=NULL;
}

void di_pre_handler(long vpid, long event_id)
{
    PUT_EVENT(vpid, event_id, START);
}

void di_post_handler(long vpid, long event_id)
{
    PUT_EVENT(vpid, event_id, END);
}

void di_fini_backend()
{
    int fd=open(tracefile, flags);
    PROCESS_TRACE(fd, funcs, buffer);
    close(fd);
}

```

Note that backend portability is determined only by the extension code, since there are no platform-specific details in the DITools interface. Platform-dependent pieces (e.g. the callback dispatcher) are provided by the DITools runtime.

**3-Build the extension module,** by compiling the above code like a standard shared library.

```
cc -shared -o progmon.so progmon.c
```

**4-Write the configuration file** to be parsed by DITools at extension time. It should declare which backend should be loaded (1) and specify that our backend routines should be invoked as *callbacks* at every reference to dynamically-linked runtime services (2).

```
// begin of backends section
// (1) request the module "progmon.so":
BACKEND backends/DIFlow/progmon.so
// end of backends section
#commands
// begin of commands section
// (2) request the installation of
// the callback dispatcher at every
// dynamically-linked reference done
// by the MAIN module
MAIN *      DIRUNTIME callback_dispatcher
// end of commands section
```

**5-Run your unmodified program.** Specify your config file and set-up the system to load the DI runtime at program startup. This last step is system dependent, as explained in 2.2.

```
$ setenv DI_CONFIG_FILE progmon.conf
$ setenv LD_PRELOAD diruntime.so
$ <your program>
```

## 4.2 Service improvement

In this example, we replace the definition of FFT used by a program, by another one that makes a more efficient computation exploiting multiple CPUs. Given the new FFT service, the extension should simply bridge differences in the interface:

```
void fft_bridge(float *data, long size)
{
    n = get_num_processors();
    new_data = reshape_data(n, data);
    spawn(n, parallel_fft, new_data, size);
    wait_for_end();
}
```

This routine computes the available number of processors, prepares the data to be used by the threads, and spawns threads to execute a parallel FFT using the reshaped data. Once built, we should simply override the old FFT service using the redefinition facility, either through the config file or at run-time (di\_rebind).

## 4.3 Filesystem extension

The third example is a filesystem extension that allows applications to transparently access remote

files. Requests corresponding to remote filesystems are redirected to a server running elsewhere. The extension should provide wrappers for filesystem services, analyze the arguments and then choose which underlying service should be invoked to complete the request.

Usually, dynamically-linked programs do not trap directly to the operating system for reading and writing. Traps are usually encapsulated in a system library (e.g. libc) and are exported as library functions. Therefore, DITools can interpose the above routines to these functions to achieve the expected behavior, without the need of system-call redirection functionality.

```
int fs_write(int fd, char *b, int s)
{
    if remote_fd(fd) {
        a = marshal(b,s);
        r = send_request(server, WRITE, a);
    } else
        r = base_fs_write(fd,b,s);
    return r;
}
```

## 4.4 Data stream processing

In this last example, we illustrate how to extend I/O services to encrypt and decrypt the data stream. Given the appropriate encryption algorithm, the extension should simply provide the routines that combine in the right order the encryption/decryption process and the I/O operation:

```
int crypt_read(int fd, char *b, int len)
{
    int r;
    r = read(fd, b, len);
    if (encrypted_channel(fd))
        decrypt(b, r);
    return(r);
}

int crypt_write(int fd, char *b, int len)
{
    int r;
    if (encrypted_channel(fd))
        crypt(b, len);
    r = write(fd, b, len);
    return(r);
}
```

These routines can be used to redefine previous read/write services. To make encryption transparent to the program, channels to be encrypted can be determined by the extension. For example, it can decide to encrypt only those data streams that are sent or received from the network.

## 5 Related work

In software environments that are built from separate modules, interactions frequently happen to be limited to well-defined interfaces between modules. Being able to interpose functionality in the midst of these boundaries, preserving physical encapsulation, has been recognized as very convenient from the extensibility point of view. This has motivated many different approaches to the problem, which are reviewed in this section.

In the first place, many systems provide interposition facilities for the system-call interface. Classical microkernels like Mach[1] or recent kernels like Pebble[7] are examples of it. It is possible to use these facilities to enable profiling (like strace in Linux) as well as functionality extension (like UFO[2]). The usefulness of this kind of interposition has been the motivation for developing extension-enabling frameworks using system-call redirection, like COLA[14], Interposition Agents[13] or SLIC[9].

Our approach differs from the above extension-enabling frameworks in that we do not rely on operating-system provided interposition services. Moreover, we are providing these facilities for interfaces between user-level modules.

Many of the techniques used inside operating-system kernels in order to dynamically accommodate functionality (like device drivers, stackable file systems [11] or other kernel modules) are working examples of extensibility and flexible composition. However, these techniques are focused to specific kernel interfaces, and its use is typically restricted to privileged users. There is a considerable amount of research on building *extensible systems*, a kind of systems that would allow generic, safe, and application-specific extensions (e.g. SPIN[5], VINO[16]), although this research is biased towards safety concerns of in-kernel extensions.

In contrast to the above techniques, we are taking a

generic approach. Our goal is to enable extensions to be attached at any available interface, by providing convenient binding mechanisms for these extensions. In addition, we are looking at backwards-compatible ways of extending the execution environment of today programs, without new kernel services.

Finally, there are also approaches to structured dynamic extension at user-level procedure-call boundaries. We can find it within advanced runtime environments (also known as “component platforms”), like ORB implementations [8] and COM+ [21]. These platforms allow subclassing of binary components, and their mechanisms (cross-component inheritance, containment or aggregation) can be seen as specialized forms of interposition. These platforms aim to provide component interoperability.

DITools derives its extension and dynamic loading abilities directly from the system’s ABI, exploiting physical encapsulation, and being largely independent of language and program development issues. In contrast, component platforms are tied to language encapsulation, and they usually need to enforce boundaries using object-oriented programming and interface definition languages. They are neither appropriate to extend nor to customize the execution environment of each and every program that runs in a given system.

We found also tools oriented to enable interposition aside of any object middleware, like Detours for Win32 [12] and SLI for Solaris [6]. In Detours, function calls are dynamically intercepted by rewriting function images in order to redirect the control flow to different locations. In contrast, SLI interposition is based on symbol preemption in the resolution mechanism. In this way, SLI can dynamically introduce profiling and tracing functionality into dynamically-linked programs without changing the program image. Last releases of Solaris include support for modifying bindings done by the runtime linker [19].

Tools like Detours and SLI are similar to DITools in that they work at application-level and provide similar interposition facilities. However, they exhibit limitations on their abilities to control definitions and bindings that do not exist in our framework. In particular, we allow selective rebinding of references, while these tools can only do global redefinitions. Detours requires altering the text pages and performs very intrusive changes in the program



(e.g. it changes memory references for the original definition). Symbols redefined by SLI will persist until the program exits. On the other hand, the Solaris runtime-linker support is a platform-specific approach that focuses on process monitoring.

## 6 Conclusions

In this article, we describe DITools, an infrastructure devoted to extend applications at run time, and to tune execution environments by dynamically changing bindings between binary modules.

DITools works at application level, and does not require superuser intervention. Moreover, it uses standard object file formats and common tools, and preserves backwards compatibility. We believe that DITools demonstrates to which point current loading and execution services can be improved to ease the job of researchers, developers and users.

We have also evaluated the overhead of DITools. The evaluation shows that it depends on the dynamic usage of extensions. We believe that this overhead can be tolerated in the most common uses of this tool.

DITools is currently being used for monitoring and trace collection purposes, as well as to build research prototypes for scheduling and I/O research. A distribution of the tools is available for research and academic purposes. Please consult <http://www.ac.upc.es/recerca/CAP/DITools>.

## 7 Acknowledgments

The authors would like to thank the NANOS project, the CEPBA-UPC team, and the I/O group, for performing the testing of the tools, as well as for their valuable feedback. Specially Xavier Martorell with its CPU manager, and Jordi Caubet with MPTrace. We also thank the GSOMK people for encouraging this research, and the CEPBA-UPC staff for its help during the evaluation.

Finally, we would also like to thank the anonymous referees for their insightful comments, as well as the USENIX shepherd Liuba Shrira for its help during the revision of the paper.

## References

- [1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young "Mach: A New Kernel Foundation for UNIX Development", In Proc. of the Summer USENIX Conf., Aug. 1986
- [2] A. Alexandrov, M. Ibel, K. Schauser, C. Scheiman "Ufo: A Personal Global File System Based on User-Level Extensions to the Operating System", ACM TOCS, Aug. 1998
- [3] J. Arnold "ELF: An Object File to Mitigate Mischievous Misoneism", In Proc. of the Summer USENIX Conference, 1990
- [4] AT&T "System V Application Binary Interface", Published by Prentice-Hall, 1990
- [5] B. Bershad et al. "Extensibility, Safety and Performance in the SPIN Operating System", In Proc. of the 15th SOSP, 1995
- [6] T. Curry "Profiling and Tracing Dynamic Library Usage Via Interposition", In Proc. of the USENIX Summer Technical Conference, 1994
- [7] E. Gabber, C. Small, J. Bruno, J. Brustoloni, A. Silberschatz "The Pebble Component-Based Operating System", In Proc. of the USENIX Annual Technical Conference, June 1999
- [8] O. Gampel, A. Gregor, S. B. Hassen, D. Johnson, N. Jnsson, D. Racioppo, H. Stlinger, K. Washida, L. Widengren "IBM Component Broker Connector Overview", Document Number SG24-2022-02, IBM Corp. 1998
- [9] D. Ghormley, S. Rodrigues, D. Petrou, T. Anderson "SLIC: An Extensibility System for Commodity Operating Systems", In Proc. of the USENIX Annual Technical Conference, June 1998
- [10] M. Gonzalez, A. Serra, X. Martorell, J. Oliver, E. Ayguade, J. Labarta, N. Navarro "Applying Interposition Techniques for Performance Analysis of OpenMP Applications", In Proc. of IPDPS'00, May 2000
- [11] J. Heidemann, Popek "File System Development with Stackable Layers", In Proc. of ACM TOCS, Feb. 1994

- [12] G. Hunt, D. Brubacher "Detours: Binary Interception of Win32 Functions", Microsoft Research, Technical Report MSR-TR-98-33, Feb. 1999
- [13] M. Jones "Interposition Agents: Transparently Interposing User Code at the System Interface", In Proc. of the 14th ACM Symposium on Operating System Principles, Dec. 1993
- [14] E. Krell, B. Krishnamurthy "COLA: Customized Overlaying", In Proc. of the Winter USENIX Conference, 1992
- [15] P. Pardyak, B. Bershad "Dynamic Binding for an Extensible System", 2nd OSDI Symposium, Oct. 1996
- [16] M. Seltzer, Y. Endo, C. Small, K. Smith "Dealing with disaster: Surviving misbehaved kernel abstractions", 2nd OSDI Symposium, Oct. 1996
- [17] A. Serra, X. Martorell, N. Navarro "Dynamically-linking Extensions and the Memoization Experience", Technical Report UPC-DAC-1999-17, 1999
- [18] A. Serra, N. Navarro "Extending the Execution Environment with DITools", Technical Report UPC-DAC-1999-26, 1999
- [19] Sun Microsystems Inc. "Linker and Libraries Guide", Solaris 2.6 Software Developer Collection, 1997
- [20] D. Orr, R. Mecklenburg "OMOS - An Object Server for Program Execution", In Proc. of the IWOOS, 1992
- [21] D. Platt "Understanding COM+", Microsoft Press, 1999
- [22] Z. Xu, B. Miller, O. Naim "Dynamic Instrumentation of Threaded Applications", In Proc. of 7th ACM SIGPLAN, May 1999

# Portable Multithreading

## The Signal Stack Trick For User-Space Thread Creation

Ralf S. Engelschall

*Technische Universität München (TUM)*

rse@engelschall.com, <http://www.engelschall.com>

### Abstract

This paper describes a pragmatic but portable fallback approach for creating and dispatching between the machine contexts of multiple threads of execution on Unix systems that lack a dedicated user-space context switching facility. Such a fallback approach for implementing machine contexts is a vital part of a user-space multithreading environment, if it has to achieve maximum portability across a wide range of Unix flavors. The approach is entirely based on standard Unix system facilities and ANSI-C language features and especially does not require any assembly code or platform specific tricks at all. The most interesting issue is the technique of creating the machine context for threads, which this paper explains in detail. The described approach closely follows the algorithm as implemented by the author for the popular user-space multithreading library *GNU Portable Threads (GNU Pth, [25])* which this way quickly gained the status of one of the most portable user-space multithreading libraries.

## 1 Introduction

### 1.1 Multithreading

The paradigm of programming with multiple threads of execution (aka *multithreading*) is already a very old one and dates back to the decades of programming with *co-routines* [2, 3]. Paradoxically, the use of threads on Unix platforms did not become popular until the early 1990s.

#### Multithreading Advantages

Multithreading can provide many benefits for applications (good runtime concurrency, parallel programming techniques can be implemented more easily, the popular procedural programming style can be combined with multiple threads of execution, *etc.*) but the most interesting ones are usually performance gains and reduced resource consumption. Because in contrast to multiprocess applications, multithreaded ones usually require less system resources (mainly memory) and their internal communication part can leverage from the shared address space.

#### Multithreading and Applications

Nevertheless there still exist just a few real applications in the free software world that use multithreading for their benefit, although their application domains are predestined for multithreading. For instance, the popular Apache webserver as of version 1.3 still uses a pre-forking process model for serving HTTP requests, although two experiments with multithreaded Apache variants in 1996 (with *rsthreads* [27]) and 1998 (with *NSPR* [31]) already showed great performance boosts. The same applies to many similar applications.

The reason for this restraint mainly is that for a long time, multithreading facilities under Unix were rare. The situation became better after some vendors like *Sun* and *DEC* incorporated threading facilities into their Unix flavors and *POSIX* standardized a threading *Application Programming Interface (API)* (aka *Pthreads* [1]). But an API and a few vendor implementations are not enough to fulfill the portability requirements of modern free software packages. Here stand-alone and really portable multithreading environments are needed.

The author collected and evaluated over twenty (mostly user-space) available multithreading facilities for Unix systems (see Table 1), but only a few of them are freely available and showed to be really portable. And even the mostly portable ones suffered from the fact that they partly depend on assembly code or platform specific tricks usually related to the creation and dispatching of the individual threads. This means that the number of platforms they support is limited and applications which are based on these facilities are only portable to those platforms. This situation is not satisfactory, so application authors still avoid the use of multithreading if they want to (or have to) achieve maximum portability for their application.

A pragmatic and mostly portable fallback technique for implementing user-space threads can facilitate wider use of multithreading in free software applications.

#### Ingredients of a Thread

A Unix process has many ingredients, but the most important ones are its memory mapping table, the signal

Package	Genesis	Latest Version	Implementation Space	Thread Mapping	Active Development	Experimental State	Open Source	Pthread API	Pthread Shared Memory	Native API	Native API > Pthread API	Preemptive Scheduling	Portability	Assembly Code	SysCall Wrap.	
gnu-pth	1999	1.3.5	user	n:1	yes	no	yes	yes	no	yes	yes	no	no	full/mcsc+sjlj	no	partly
cmu-lwp	1984	1.4	user	n:1	yes	no	yes	no	-	yes	yes	partly	no	semi/fixed:8	yes	no
fsu-pthread	1992	3.13	user	n:1	no	no	yes	yes	no	no	-	-	yes	semi/fixed:6	yes	yes
mit-pthread	1993	1.8.9	user	n:1	no	no	yes	yes	no	no	-	-	yes	semi/fixed:17	yes	yes
ptl	1997	990622	user	n:1	no	no	yes	yes	no	no	-	-	yes	semi/fixed:10	yes	yes
linuxthreads	1997	2.1.2	user+kernel	1:1	yes	no	yes	yes	no	no	-	-	yes	semi/fixed:5	yes	yes
uthread	1998	3.4	user	n:1	yes	no	yes	yes	no	no	-	-	yes	semi/fixed:2	yes	yes
cthread	1991	991115	user	n:1	no	no	yes	no	-	yes	yes	no	no	semi/fixed:8	yes	yes
openthreads/qt	1996	2.0	user	n:1	no	no	yes	no	-	yes	no	no	no	semi/fixed:9	yes	no
rt++/qt	1996	1.0	user	n:1	no	no	yes	no	-	yes	yes	no	no	semi/fixed:9	yes	no
rsthreads	1996	980331	user	n:1	no	yes	yes	no	-	yes	no	no	no	semi/fixed:9	yes	no
pcthread	1996	1.0	user	n:1	no	yes	yes	yes	no	no	-	-	yes	semi/fixed:1	yes	no
bbthreads	1996	0.3	kernel	1:1	no	yes	yes	no	-	yes	no	-	yes	semi/fixed:1	yes	no
jkthreads	1998	1.2	kernel	1:1	no	yes	yes	no	-	yes	no	-	yes	semi/fixed:1	yes	no
nthreads	1997	970604	user	n:1	no	yes	yes	no	-	yes	no	-	no	semi/fixed:9	yes	partly
rethreads	1993	930614	user	n:1	no	yes	yes	no	-	yes	no	-	no	semi/fixed:4	yes	no
coro	1999	1.0.3	user	n:1	no	yes	yes	no	-	yes	no	-	no	semi/fixed:1	yes	no
greenthreads	1995	1.2	user	n:1	no	no	no	no	-	yes	yes	-	yes	full/mcsc	no	no
solaris-pthread	NN	2.7	user+kernel	n:m	yes	no	no	yes	yes	yes	yes	no	yes	NN	NN	yes
tru64-pthread	NN	5.0	user+kernel	n:m	yes	no	no	yes	yes	no	no	no	yes	NN	NN	yes
aix-pthread	NN	4.3	user+kernel	1:1	yes	no	no	yes	yes	no	no	no	yes	NN	NN	yes

**Table 1:** Summary of evaluated multithreading packages and some of their determined characteristics. Notice that mostly all packages contain assembly code and are just semi-portable, i.e., they support only a fixed set of platforms and do not automatically adjust for new ones.

dispatching table, the signal mask, the set of file descriptors and the machine context. The machine context in turn consists of at least the CPU registers including the program counter and the stack pointer. In addition, there can be light-weight processes (LWP) or threads, which usually share all attributes with the underlying (heavy-weight) process except for the machine context.

### Kernel-Space vs. User-Space

Those LWPs or threads, on a Unix platform classically can be implemented either in kernel-space or in user-space. When implemented in kernel-space, one usually calls them LWPs or kernel threads, otherwise (user-space) threads. If threads are implemented by the kernel, the thread context switches are performed by the kernel without notice by the application, similar to the dispatching of processes. If threads are implemented in user-space, the thread context switches are performed usually by an application library without notice by the kernel. Additionally, there exist hybrid threading approaches, where typically a user-space library binds one or more user-space threads to one or more kernel-space LWPs.

### Thread Models

The vendor threading facilities under *Sun Solaris*, *IBM AIX*, *DEC Tru64* (formerly *DIGITAL UNIX* or *OSF/1*) and *SGI IRIX* use a **M:N** mapping [21, 30], i.e., **M** user-

space threads are mapped onto **N** kernel-space LWPs. On the other hand, *LinuxThreads* [29] under *GNU/Linux* uses a **1:1** mapping and pure user-space implementations like *GNU Pth*, *FSU pthreads* or *MIT pthreads*, etc. use a **M:1** mapping [25, 22, 23].

From now on we focus on such **M:1** user space threading approaches, where one or more user space threads are implemented inside a single kernel space process. The exercise is to implement this by using standardized Unix system and ANSI-C language facilities *only*.

## 1.2 The Exercise

As we have mentioned, a thread shares its state with the underlying process except for the machine context. So the major task for a user-space threading system is to create and dispatch those machine contexts.

In practice, the second major task it has to do is to ensure that no thread by accident blocks the whole process (and thereby all other threads). Instead when an operation would block, the threading library should suspend only the execution of the current thread and in the meantime dispatch the remaining threads. But this task is outside the scope of this paper (see [11] for details about this task). We focus only on the aspect of machine context handling.



### 1.3 The Curse of Portability

Our goal of real portability for a threading system causes some non-trivial problems which have to be solved. The most obvious one is that dealing with machine contexts usually suffers from portability, because it is a highly CPU dependent task for which not every Unix flavor provides a standardized API. Although such an API would be not too hard for vendors to provide, because in principle it is just a matter of switching a few CPU registers (mainly the program counter and the stack pointer).

#### Assembly Code Considered Harmful

Additionally, we disallow the use of any assembly solutions or platform specific tricks, because then the threading system again would be only semi-portable, *i.e.*, it can be ported to  $N$  platforms but on the  $(N+1)$ th platform one has to manually adjust or even extend it to work there, too.

This is usually not acceptable, even if it also makes solving the problems harder. At least most of the known free software user-space threading systems [22, 23, 24] do not restrict themselves to this and therefore are just semi-portable. But real portability should be a major goal.

## 2 Problem Analysis

### 2.1 The Task in Detail

Our task is simple in principle: provide an API and corresponding implementation for creating and dispatching machine contexts on which user-space threads can be implemented.

#### The Proposed API

In detail we propose the following *Application Programmers Interface* (API) for the machine context handling:

- A data structure of type `mctx_t` which holds the machine context.
- A function “`void mctx_create(mctx_t *mctx, void (*sf_addr) (void *), void *sf_arg, void *sk_addr, size_t sk_size)`” which creates and initializes a machine context structure in `mctx` with a start function `sf_addr`, a start function argument `sf_arg`, and a stack starting at `sk_addr`, which is `sk_size` bytes in size.
- A function “`void mctx_save(mctx_t *mctx)`” which saves the current machine context into the machine context structure `mctx`.
- A function “`void mctx_restore(mctx_t *mctx)`” which restores the new machine context from the machine context structure `mctx`. This

function does not return to the caller. Instead it does return at the location stored in `mctx` (which is either `sf_addr` from a previous `mctx_create` call or the location of a previous `mctx_save` call).

- A function “`void mctx_switch(mctx_t *mctx_old, mctx_t *mctx_new)`” which switches from the current machine context (saved to `mctx_old` for later use) to a new context (restored from `mctx_new`). This function returns only to the caller if `mctx_restore` or `mctx_switch` is again used on `mctx_old`.

### 2.2 Technical Possibilities

Poking around in the references of the ANSI-C language reference and the Unix standards show the following functions on which an implementation can be based:

- There is the `ucontext(3)` facility with the functions `getcontext(3)`, `makecontext(3)`, `swapcontext(3)` and `setcontext(3)` which conform to the *Single Unix Specification*, Version 2 (*SUSv2* [20], aka *Unix95/98*). Unfortunately these are available on modern Unix platforms only.
- There are the `jmp_buf` based functions `setjmp(3)` and `longjmp(3)` which conform to ISO 9899:1990 (ISO-C) and the `sigjmp_buf` based `sigsetjmp(3)` and `siglongjmp(3)` functions which conform to IEEE Std1003.1-1988 (*POSIX*), and *Single Unix Specification*, Version 2 (*SUSv2* [20], aka *Unix95/98*). The first two functions are available really on all Unix platforms, the last two are available only on some of them.

On some platforms `setjmp(3)` and `longjmp(3)` save and restore also the signal mask (if one does not want this semantics, one has to call `_setjmp(3)` and `_longjmp(3)` there) while on others one has to explicitly use the superset functions `sigsetjmp(3)` and `siglongjmp(3)` for this. In our discussion we can assume that `setjmp(3)` and `longjmp(3)` save and restore the signal mask, because if this is not the case in practice, one easily can replace them with `sigsetjmp(3)` and `siglongjmp(3)` calls (if available) or (if not available) emulate the missing functionality manually with additional `sigprocmask(2)` calls (see `pth_mctx.c` in *GNU Pth* [25]).

- There is the function `sigaltstack(2)` which conforms to the *Single Unix Specification*, Version 2 (*SUSv2* [20], aka *Unix95/98*) and its ancestor function `sigstack(2)` from *4.2BSD*. The

last one exists only on *BSD*-derived platforms, but the first function already exists on all current Unix platforms.

## 2.3 Maximum Portability Solution

The maximum portable solution obviously is to use the standardized `makecontext(3)` function to create threads and `switchcontext(3)` or `getcontext(3)/setcontext(3)` to dispatch them. And actually these are the preferred functions modern user-space multithreading systems are using. We could easily implement our proposed API as following (all error checks omitted for better readability):

```
/* machine context data structure */
typedef struct mctx_st {
    ucontext_t uc;
} mctx_t;

/* save machine context */
#define mctx_save(mctx) \
    (void) getcontext (&(mctx)->uc)

/* restore machine context */
#define mctx_restore(mctx) \
    (void) setcontext (&(mctx)->uc)

/* switch machine context */
#define mctx_switch(mctx_old, mctx_new) \
    (void) swapcontext (&(mctx_old)->uc, \
        &(mctx_new)->uc)

/* create machine context */
void mctx_create(
    mctx_t *mctx,
    void (*sf_addr)(void *), void *sf_arg,
    void *sk_addr, size_t sk_size)
{
    /* fetch current context */
    getcontext(&(mctx->uc));

    /* adjust to new context */
    mctx->uc.uc_link = NULL;
    mctx->uc.uc_stack.ss_sp = sk_addr;
    mctx->uc.uc_stack.ss_size = sk_size;
    mctx->uc.uc_stack.ss_flags = 0;

    /* make new context */
    makecontext(&(mctx->uc),
        sf_addr, 1, sf_arg);
    return;
}
```

Unfortunately there are still lots of Unix platforms where this approach cannot be used, because the standardized `ucontext(3)` API is not provided by the vendor. Actually the platform test results for *GNU Pth* (see Table 2 below) showed that only 7 of 21 successfully tested Unix flavors provided the standardized API (`makecontext(3)`, etc.). On all other platforms, *GNU Pth* was forced to use the fallback approach of implementing the machine context as we will describe in the

following. Obviously this fallback approach has to use the remaining technical possibilities (`sigsetjmp(3)`, etc.).

Operating System	Architecture(s)	mcsc	sjlj
FreeBSD 2.x/3.x	Intel	no	yes
FreeBSD 3.x	Intel, Alpha	no	yes
NetBSD 1.3/1.4	Intel, PPC, M68K	no	yes
OpenBSD 2.5/2.6	Intel, SPARC	no	yes
BSDI 4.0	Intel	no	yes
Linux 2.0.x glibc 1.x/2.0	Intel, SPARC, PPC	no	yes
Linux 2.2.x glibc 2.0/2.1	Intel, Alpha, ARM	no	yes
Sun SunOS 4.1.x	SPARC	no	yes
Sun Solaris 2.5/2.6/2.7	SPARC	yes	yes
SCO UnixWare 2.x/7.x	Intel	yes	yes
SCO OpenServer 5.0.x	Intel	no	yes
IBM AIX 4.1/4.2/4.3	RS6000, PPC	yes	yes
HP HP/UX 9.10/10.20	HPPA	no	yes
HP HP/UX 11.0	HPPA	yes	yes
SGI IRIX 5.3	MIPS 32/64	no	yes
SGI IRIX 6.2/6.5	MIPS 32/64	yes	yes
ISC 4.0	Intel	no	yes
Apple MacOS X	PPC	no	yes
DEC OSF1/Tru64 4.0/5.0	Alpha	yes	yes
SNI ReliantUNIX	MIPS	yes	yes
AmigaOS	M68K	no	yes

**Table 2:** Summary of operating system support. The level and type of support found on each tested operating system. mcsc: functional `makecontext(3)/switchcontext(3)`, sjlj: functional `setjmp(3)/longjmp(3)` or `sigsetjmp(3)/siglongjmp(3)`. See file *PORTING* in *GNU Pth* [25] for more details.

## 2.4 Remaining Possibilities

Our problem can be divided into two parts, an easy one and a difficult one.

### The Easy Part

That `setjmp(3)` and `longjmp(3)` can be used to implement user-space threads is commonly known [24, 27, 28]. Mostly all older portable user-space threading libraries are based on them, although some problems are known with these facilities (see below). So it becomes clear that we also have to use these functions and base our machine context (`mctx_t`) on their `jmp_buf` data structure.

We immediately recognize that this way we have at least solved the dispatching problem, because our `mctx.save`, `mctx.restore` and `mctx.switch` functions can be easily implemented with `setjmp(3)` and `longjmp(3)`.

### The Difficult Part

Nevertheless, the difficult problem of how to create the machine context remains. Even knowing that our machine context is `jmp_buf` based is no advantage to us. A `jmp_buf` has to be treated by us as an opaque data structure — for portability reasons. The only operations we can perform on it are `setjmp(3)` and `longjmp(3)` calls,

of course. Additionally, we are forced to use `sigaltstack(3)` for our stack manipulations, because it is the only portable function which actually deals with stacks.

So it is clear that our implementation for `mctx_create` has to play a few tricks to use a `jmp_buf` for passing execution control to an arbitrary startup routine. And our approach has to be careful to ensure that it does not suffer from unexpected side-effects. It should be also obvious that we cannot again expect to find an easy solution (as for `mctx_save`, `mctx_restore` and `mctx_switch`), because `setjmp(3)` and `sigaltstack(3)` cannot be trivially combined to form `mctx_create`.

### 3 Implementation

As we have already discussed, our implementation contains an easy part (`mctx_save`, `mctx_restore` and `mctx_switch`) and a difficult part (`mctx_create`). Let us start with the easy part, whose implementation is obvious (all error checks again omitted for better readability):

```
/* machine context data structure */
typedef struct mctx_st {
    jmp_buf jrb;
} mctx_t;

/* save machine context */
#define mctx_save(mctx) \
    (void) setjmp((mctx)->jrb)

/* restore machine context */
#define mctx_restore(mctx) \
    longjmp((mctx)->jrb, 1)

/* switch machine context */
#define mctx_switch(mctx_old, mctx_new) \
    if (setjmp((mctx_old)->jrb) == 0) \
        longjmp((mctx_new)->jrb, 1)

/* create machine context */
void mctx_create(
    mctx_t *mctx,
    void (*sf_addr)(void *), void *sf_arg,
    void *sk_addr, size_t sk_size)
{
    ...initialization of mctx to be filled in...
}
```

There is one subtle but important point we should mention: The use of the C pre-processor `#define` directive to implement `mctx_save`, `mctx_restore` and `mctx_switch` is intentional. For technical reasons related to `setjmp(3)` semantics and return related stack behavior (which we will explain later in detail) we *cannot* implement these three functions (at least not `mctx_switch`) as C functions if we want to achieve maximum portability across all platforms. Instead they have to be implemented as pre-processor macros.

### 3.1 Algorithm Overview

The general idea for `mctx_create` is to configure the given stack as a signal stack via `sigaltstack(2)`, send the current process a signal to transfer execution control onto this stack, save the machine context there via `setjmp(3)`, get rid of the signal handler scope and bootstrap into the startup routine.

The real problem in this approach comes from the signal handler scope which implies various restrictions on Unix platforms (the signal handler scope often is just a flag in the process control block (PCB) which various system calls, like `sigaltstack(2)`, check before allowing the operation – but because it is part of the process state the kernel manages, the process cannot change it itself). As we will see, we have to perform a few tricks to get rid of it. The second main problem is: how do we prepare the calling of the start routine without immediately entering it?

### 3.2 Algorithm

The input to the `mctx_create` function is the machine context structure `mctx` which should be initialized, the thread startup function address `sf_addr`, the thread startup function argument `sf_arg` and a chunk of memory starting at `sk_addr` and `sk_size` bytes in size, which should become the threads stack.

The following algorithm for `mctx_create` is directly modeled after the implemented algorithm one can find in *GNU Pth* [25], which in turn was derived from techniques originally found in *rsthreads* [27]:

1. Preserve the current signal mask and block an arbitrary worker signal (we use `SIGUSR1`, but any signal can be used for this – even an already used one). This worker signal is later temporarily required for the trampoline step.
2. Preserve a possibly existing signal action for the worker signal and configure a trampoline function as the new temporary signal action. The signal delivery is configured to occur on an alternate signal stack (see next step).
3. Preserve a possibly active alternate signal stack and configure the memory chunk starting at `sk_addr` as the new temporary alternate signal stack of length `sk_size`.
4. Save parameters for the trampoline step (`mctx`, `sf_addr`, `sf_arg`, etc.) in global variables, send the current process the worker signal, temporarily unblock it and this way allow it to be delivered on the signal stack in order to transfer execution control to the trampoline function.

5. After the trampoline function asynchronously entered, save its machine context in the *mctx* structure and immediately return from it to terminate the signal handler scope.
6. Restore the preserved alternate signal stack, preserved signal action and preserved signal mask for worker signal. This way an existing application configuration for the worker signal is restored.
7. Save the current machine context of *mctx\_create*. This allows us to return to this point after the next trampoline step.
8. Restore the previously saved machine context of the trampoline function (*mctx*) to again transfer execution control onto the alternate stack, but this time without(!) signal handler scope.
9. After reaching the trampoline function (*mctx*) again, immediately bootstrap into a clean stack frame by just calling a second function.
10. Set the new signal mask to be the same as the original signal mask which was active when *mctx\_create* was called. This is required because in the first trampoline step we usually had at least the worker signal blocked.
11. Load the passed startup information (*sf\_addr*, *sf\_arg*) from *mctx\_create* into local (stack-based) variables. This is important because their values have to be preserved in machine context dependent memory until the created machine context is the first time restored by the application.
12. Save the current machine context for later restoring by the calling application.
13. Restore the previously saved machine context of *mctx\_create* to transfer execution control back to it.
14. Return to the calling application.

When the calling application now again switches into the established machine context *mctx*, the thread starts running at routine *sf\_addr* with argument *sf\_arg*. Figure 1 illustrates the algorithm (the numbers refer to the algorithm steps listed above).

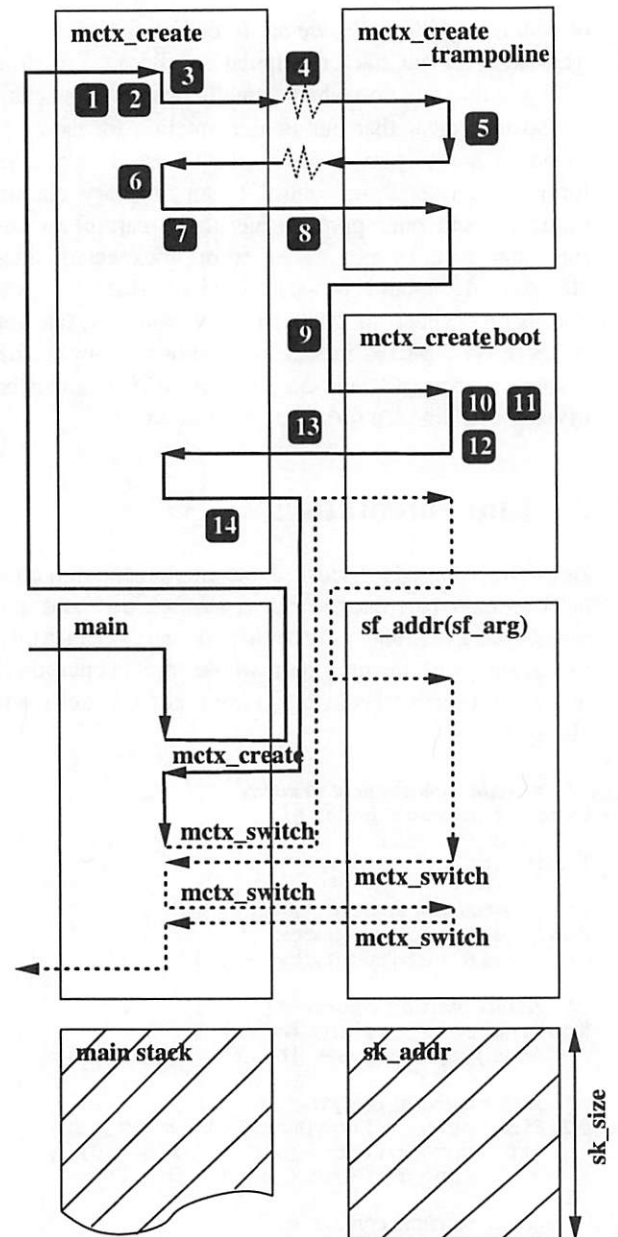


Figure 1: Illustration of the machine context creation procedure. The thick solid lines and numeric marks correspond to the algorithm steps as described in section 3.2. The thick dotted lines show a possible further processing where a few context switches are performed to dispatch between the main thread and the new created thread.

### 3.3 Source Code

The corresponding ANSI-C code, which implements *mctx\_create*, is a little bit more complicated. But with the presented algorithm in mind, it is now straightforward.

```
static mctx_t      mctx_caller;
static sig_atomic_t mctx_called;

static mctx_t      *mctx_create;
```



```

static void      (*mctx_creat_func)(void *); /* Step 9: */
static void      *mctx_creat_arg;           mctx_create_boot();
static sigset_t  mctx_creat_sigs;           }

void mctx_create(
    mctx_t *mctx,
    void (*sf_addr)(void *), void *sf_arg,
    void *sk_addr, size_t sk_size)
{
    struct sigaction sa;
    struct sigaction osa;
    struct sigaltstack ss;
    struct sigaltstack oss;
    sigset_t osigs;
    sigset_t sigs;

    /* Step 1: */
    sigemptyset(&sigs);
    sigaddset(&sigs, SIGUSR1);
    sigprocmask(SIG_BLOCK, &sigs, &osigs);

    /* Step 2: */
    memset((void *)&sa, 0,
           sizeof(struct sigaction));
    sa.sa_handler = mctx_create_trampoline;
    sa.sa_flags = SA_ONSTACK;
    sigemptyset(&sa.sa_mask);
    sigaction(SIGUSR1, &sa, &osa);

    /* Step 3: */
    ss.ss_sp = sk_addr;
    ss.ss_size = sk_size;
    ss.ss_flags = 0;
    sigaltstack(&ss, &oss);

    /* Step 4: */
    mctx_creat = mctx;
    mctx_creat_func = sf_addr;
    mctx_creat_arg = sf_arg;
    mctx_creat_sigs = osigs;
    mctx_called = FALSE;
    kill(getpid(), SIGUSR1);
    sigfillset(&sigs);
    sigdelset(&sigs, SIGUSR1);
    while (!mctx_called)
        sigsuspend(&sigs);

    /* Step 6: */
    sigaltstack(NULL, &ss);
    ss.ss_flags = SS_DISABLE;
    sigaltstack(&ss, NULL);
    if (!(oss.ss_flags & SS_DISABLE))
        sigaltstack(&oss, NULL);
    sigaction(SIGUSR1, &osa, NULL);
    sigprocmask(SIG_SETMASK,
                &osigs, NULL);

    /* Step 7 & Step 8: */
    mctx_switch(&mctx_caller, mctx);

    /* Step 14: */
    return;
}

void mctx_create_trampoline(int sig)
{
    /* Step 5: */
    if (mctx_save(mctx_creat) == 0) {
        mctx_called = TRUE;
        return;
    }
}

void mctx_create_boot(void)
{
    void (*mctx_start_func)(void *);
    void *mctx_start_arg;

    /* Step 10: */
    sigprocmask(SIG_SETMASK,
                &mctx_creat_sigs, NULL);

    /* Step 11: */
    mctx_start_func = mctx_creat_func;
    mctx_start_arg = mctx_creat_arg;

    /* Step 12 & Step 13: */
    mctx_switch(mctx_creat, &mctx_caller);

    /* The thread "magically" starts... */
    mctx_start_func(mctx_start_arg);

    /* NOTREACHED */
    abort();
}

```

### 3.4 Run-time Penalty

After this discussion of the implementation details, an obviously occurring question now is what the expected run-time penalty is. That is, what does our presented machine context implementation cost compared to a `ucontext(3)` based solution. From the already discussed details we can easily guess that our complex machine context creation procedure (`mctx_create`) will be certainly noticeably slower than a solution based on a `ucontext(3)` facility.

But a wild guess is not sufficing for a reasonable statement. So we have written a *Simple Machine Context Benchmark* (SMCB [32]) which was used to compare run-time costs of the `mctx_create` and `mctx_switch` functions if once implemented through the `POSIX` `makecontext(3)`/`swapcontext(3)` functions (as shown in section 2.3), and once implemented with our based fallback implementation (for convenience reasons we directly used `sigjmp_buf`, `sigsetjmp(3)` and `siglongjmp(3)` in the benchmark, because all tested platforms provided this). The results are shown Table 3 below.

As one can derive from these evaluations, our signal stack trick to implement `mctx_create` in practice is approximately 15 times slower than the `makecontext(3)` based variant. This cost should not be neglected. On the other hand, the `sigsetjmp(3)`/`siglongjmp(3)` based `mctx_switch` performs about as good as the `swapcontext(3)` based variant (the reason why on most of the tested platforms it is even slightly faster is not known – but we guess it is related to a greater management overhead in the `ucontext(3)` fa-

cility, which is a superset of the functionality we require). Or in short: our presented fallback approach costs noticeable extra CPU cycles on thread creation time, but is as fast as the standardized solution under thread dispatching time.

10000 × mctx_context (in seconds):			
Platform	mctx	sjlj	overhead
Sun Solaris 2.6 (SPARC)	0.076	1.268	16.7
DEC Tru64 5.0 (Alpha)	0.019	0.235	12.4
SGI IRIX 6.5 (MIPS)	0.105	1.523	14.5
SCO UnixWare 7.0 (Intel)	0.204	3.827	18.8
HP HP/UX 11.0 (HPPA)	0.057	0.667	11.8
Average:			14.8

10000 × mctx_switch (in seconds):			
Platform	mctx	sjlj	overhead
Sun Solaris 2.6 (SPARC)	0.137	0.210	1.5
DEC Tru64 5.0 (Alpha)	0.034	0.022	0.6
SGI IRIX 6.5 (MIPS)	0.235	0.190	0.8
SCO UnixWare 7.0 (Intel)	0.440	0.398	0.9
HP HP/UX 11.0 (HPPA)	0.106	0.065	0.6
Average:			0.9

**Table 3:** Summary of Simple Machine Context Benchmark (SMCB, [32]). The speed of machine context creation and switching found on each tested operating system. **mctx**: functional `makecontext(3)/switchcontext(3)`, **sjlj**: functional `sigsetjmp(3)/siglongjmp(3)`. **overhead**: the overhead of using `sjlj` instead of `mctx`.

### 3.5 Remaining Issues

The presented algorithm and source code can be directly used in practice for implementing a minimal threading system or the concept of co-routines. Its big advantage is that if the operating system provides the required standardized primitives, we do not need to know anything at all about the machine we are running on — everything just works. Nevertheless, there remain a few special issues we still have to discuss.

#### The Waggly `longjmp(3)` after Return

On some platforms, `longjmp(3)` may not be called after the function which called the `setjmp(3)` returned. When this is done, the stack frame situation is not guaranteed to be in a clean and consistent state. But this is exactly the mechanism we use in order to get rid of the signal handler scope in step 5.

The only alternative would be to leave the signal handler via `longjmp(3)`, but then we would have another problem, as experience showed. For instance, ROBERT S. THAU's *Really Simple Threads (rsthreads)* [27] was ported to several platforms and was used to run an experimental multithreaded version of the Apache webserver. THAU's approach was similar to ours, but differed significantly in the way the signal handler is left. In particular, in an attempt to avoid the unsafe stack frame, it used a `longjmp(3)` call to leave the signal handler, rather than

returning from it. But this approach does not work on some SysV-derived kernels, as we already mentioned.

The problem is that these kernels do not “believe” that the code is out of the signal-handling context, until the signal handler has returned — and accordingly, refuse to allow readjustment of the signal stack until it has. But with the *rsthreads* approach, the signal handler that created the first thread never returns, and when *rsthreads* wants to create the second thread, these kernels refuse to readjust the signal stack, and we are stuck. So with portability in mind, we decided that it is better to get rid of the signal handler scope with the straight-forward “return” and instead fight the mentioned (simpler) problem of an unsafe stack frame.

Fortunately, in practice this is not as problematic as it seems, because evaluations (for *GNU Pth*) on a wide range of current Unix platforms showed that one can reach a safe stack frame again by just calling a function. That is the reason why our algorithm enters the second trampoline function in step 9.

#### The Uncooperative `longjmp(3)`

Even on operating systems which have working *POSIX* functions, our approach may theoretically still not work, because `longjmp(3)` does not cooperate. For instance, on some platforms the standard *libc* `longjmp(3)` branches to error-handling code if it detects that the caller tries to jump *up* the stack, *i.e.*, into a stack frame that has already returned.

This is usually implemented by comparing the current stack pointer to the one in the `jmp_buf` structure. That is why it is important for our algorithm to return from the signal handler and this way enter the (different) stack of the parent thread. In practice, the implementation in *GNU Pth* showed that then one no longer suffers from those uncooperative `longjmp(3)` implementations, but one should keep this point in mind when reaching even more uncooperative variants on esoteric Unix platforms. If it still occurs, one can only try to resume the operation by using a possibly existing platform-specific error handling hook.

#### Garbage at Bottom of Stacks

There is a subtle side-effect of our implementation: there remains some garbage at the bottom of each thread stack. The reason is that if a signal is delivered, the operating system pushes some state onto the stack, which is restored later, when the signal handler returns. But although we return from the signal handler, we jump in again, and this time we enter not directly at the bottom of the stack, because of the `setjmp(3)` in the trampoline function.

Since the operating system has to capture all CPU registers (including those that are ordinarily scratch registers or caller-save registers), there can be a fair amount

of memory at the bottom of the established thread stack. For some systems this can be even up to 1 KB of garbage [27]. But except for the additional memory consumption it does not hurt.

We just have to keep in mind this additional stack consumption when deciding the stack size (*sk\_size*). A reasonable stack size usually is between 16 and 32 KB. Less is neither reasonable nor always allowed (current Unix platforms usually require a stack to be at least 16 KB in size).

## Stack Overflows

There is a noticeable difference between the initial `main()` thread and the explicitly spawned threads: the initial thread runs on the standard process stack. This stack automatically can grow under Unix, while the stacks of the spawned threads are fixed in size. So stack overflows can occur for the spawned threads. This implies that the parent has to make a reasonable guess of the threads stack space requirement already at spawning time.

And there is no really portable solution to this problem, because even if the thread library's scheduler can detect the stack overflow, it cannot easily resize the stack. The reason is simply that the stack initialization goes hand in hand with the initialization of the start routine, as we discussed before. And this start routine has to be a real C function in order to *call*. But once the thread is running, there no longer exists such an entry point. So, even if the scheduler would be able to give the thread a new enlarged stack, there is no chance to restart the thread on this new stack.

Or more correct, there is no *portable* way to achieve it. As with the previous problems, there is a non-portable solution. That is why our implementation did not deal with this issue. Instead in practice one usually lets the scheduler just detect the stack overflow and terminate the thread. This is done by using a red zone at the top of the stack which is marked with a magic value the scheduler checks between thread dispatching operations.

Resizing solutions are only possible in semi-portable ways. One approach is to place the thread stacks into a memory mapped area (see `mmap(2)`) of the process address space and let the scheduler catch `SIGSEGV` signals. When such a signal occurs, because of a stack overflow in this area, the scheduler explicitly resizes the memory mapped area. This resizing can be done either by copying the stack contents into a new larger area which is then re-mapped to the old address or via an even more elegant way, as the vendor threading libraries of *Sun Solaris*, *FreeBSD* and *DEC Tru64* do it: the thread stacks are allocated inside memory mapped areas which are already initially a few MB in (virtual) size and then one just relies on the virtual memory sys-

tem's feature that only the actually consumed memory space is mapped.

## Startup Routine Termination

There is a cruel `abort(3)` call at the end of our `mctx_create_boot` function. This means, if the startup routine would return, the process is aborted. That is obviously not reasonable, so why have we written it this way?

If the thread returns from the startup routine, it should be cleanly terminated. But it cannot terminate itself (for instance, because it cannot free its own stack while running on it, *etc.*). So the termination handling actually is the task of the thread library scheduler. As a consequence, the thread spawning function of a thread library should be not directly `mctx_create`.

Instead the thread spawning function should use an additional trampoline function as the higher-level startup routine. And this trampoline function performs a context switch back into the thread library scheduler before the lower-level startup routine would return. The scheduler then can safely remove the thread and its machine context. That is why the `abort(3)` call is never reached in practice (more details can be found in the implementations of `pth_spawn` and `pth_exit` in `pth_lib.c` of *GNU Pth* [25])

## The sigstack(2) Fallback Situation

Not all platforms provide the standardized `sigaltstack(2)`. Instead they at least provide the *4.2BSD* ancestor function `sigstack(2)`. But one cannot trivially replace `sigaltstack(2)` by `sigstack(2)` in this situation, because in contrast to `sigaltstack(2)`, the old `sigstack(2)` does not automatically handle the machine dependent direction of stack growth.

Instead, the caller has to know the direction and always call `sigstack(2)` with the address of the bottom of the stack. So, in a real-world implementation one first has to determine the direction of stack growth in order to use `sigstack(2)` as a replacement for `sigaltstack(2)`. Fortunately this is easier than it seems on the first look (for details see the macros `AC_CHECK_STACKGROWTH` and `AC_CHECK_STACKSETUP` in file `aclocal.m4` from *GNU Pth* [25]). Alternatively if one can afford to waste memory, one can use an elegant trick: to set up a stack of size  $N$ , one allocates a chunk of memory (starting at address  $A$ ) of size  $N \times 2$  and then calls `sigstack(2)` with the parameters `sk_addr=A + (N)` and `sk_size=N`, *i.e.*, one specifies the middle of the memory chunk as the stack base.

## The Blind Alley of Brain-Dead Platforms

The world would not be as funny as it is, if really all Unix platforms would be fair to us. Instead, currently



at least one platform exists which plays unfair: unfortunately, ancient versions of the popular *GNU/Linux*. Although we will discover that it both provides `sigaltstack(2)` and `sigstack(2)`, our approach won't work on *Linux* kernels prior to version 2.2 and *glibc* prior to version 2.1.

Why? Because its *libc* provides only stubs of these functions which always return just `-1` with `errno` set to `ENOSYS`. So, this definitely means that our nifty algorithm is useless there, because its central point is `sigaltstack(2)/sigstack(2)`. Nevertheless we do not need to give up. At least not, if we, for a single brain-dead platform, accept to break our general goal of not using any platform dependent code.

So, what can we actually do here? All we have to do, is to fiddle around a little bit with the machine-dependent `jmp_buf` ingredients (by poking around in `setjmp.h` or by disassembling `longjmp(3)` in the debugger). Usually one just has to do a `setjmp(3)` to get an initial state in the `jmp_buf` structure and then manually adjust two of its fields: the program counter (usually a structure member with "pc" in the name) and the stack pointer (usually a structure member with "sp" in the name).

That is all and can be acceptable for a real-world implementation which really wants to cover mostly *all* platforms – at least as long as the special treatment is needed just for one or two platforms. But one has to keep in mind that it at least breaks one of the initial goals and has to be treated as a last chance solution.

### Functions `sigsetjmp(3)` and `siglongjmp(3)`

One certainly wants the *POSIX* thread semantics where a thread has its own signal mask. As already mentioned, on some platforms `setjmp(3)` and `longjmp(3)` do not provide this and instead one has to explicitly call `sigsetjmp(3)` and `siglongjmp(3)` instead. There is only one snare: on some platforms `sigsetjmp(3)/siglongjmp(3)` save also information about the alternate signals stack. So here one has to make sure that although the thread dispatching later uses `sigsetjmp(3)/siglongjmp(3)`, the thread creation step in `mctx_create` still uses plain `setjmp(3)/longjmp(3)` calls for the trampoline trick. One just has to be careful because the `jmp_buf` and `sigjmp_buf` structures cannot be mixed between calls to the `sigsetjmp(3)/siglongjmp(3)` and `setjmp(3)/longjmp(3)`.

### More Machine Context Ingredients

Finally, for a real-world threading implementation one usually want to put more state into the machine context structure `mctx_t`. For instance to fulfill more *POSIX* threading semantics, it is reasonable to also save and restore the global `errno` variable. All this can be easily achieved by extending the `mctx_t` structure

with additional fields and by making the `mctx_save`, `mctx_restore` and `mctx_switch` functions to be aware of them.

## 3.6 Related Work

Beside *GNU Pth* [25], there are other multithreading libraries which use variants of the presented approach for implementing machine contexts in user-space. Most notably there are ROBERT S. THAU's *Really Simple Threads* (*rsthreads*, [27]) package which uses `sigaltstack(2)` in a very similar way for thread creation, and KOTA ABE's *Portable Thread Library* (*PTL*, [24]) which uses a `sigstack(2)` approach. But because their approaches handle the signal handler scope differently, they are not able to achieve the same amount of portability and this way do not work for instance on some System-V-derived platforms.

## 3.7 Summary & Availability

We have presented a pragmatic and mostly portable fallback approach for implementing the machine context for user-space threads, based entirely on Unix system and ANSI-C language facilities. The approach was successfully tested in practice on a wide range of Unix flavors by *GNU Pth* and should also adapt to the remaining Unix platforms as long as they adhere to the relevant standards.

The *GNU Pth* package is distributed under the GNU Library General Public License (LGPL 2.1) and freely available from <http://www.gnu.org/software/pth/> and <ftp://ftp.gnu.org/gnu/pth/>.

## 3.8 Acknowledgements

I would like to thank ROBERT S. THAU, DAVID BUTENHOF, MARTIN KRAEMER, ERIC NEWTON and BRUNO HAIBLE for their comments which helped to write the initial version of this paper. Additionally, credit has to be given to CHRISTOPHER SMALL and the USENIX reviewers for their invaluable feedback which allowed this paper to be extended, cleaned up and finally published at the USENIX Annual Technical Conference 2000. Finally, thanks go to all users of *GNU Pth* for their feedback on the implementation, which helped in fine-tuning the presented approach. [rse]

## References

- [1] *POSIX 1003.1c Threading*, IEEE POSIX 1003.1c-1995, ISO/IEC 9945-1:1996
- [2] M.E. CONWAY: *Design of a separable transition-diagram compiler*, Comm. ACM 6:7, 1963, p.396-408



- [3] E.W. DIJKSTRA: *Co-operating sequential processes*, in F. Genuys (Ed.), *Programming Languages*, NATO Advanced Study Institute, Academic Press, London, 1965, p.42-112.
- [4] B. NICHOLS, D. BUTTLAR, J.P. FARREL: *Pthreads Programming - A POSIX Standard for Better Multiprocessing*, O'Reilly, 1996; ISBN 1-56592-115-1
- [5] B. LEWIS, D. J. BERG: *Threads Primer - A Guide To Multithreaded Programming*, Prentice Hall, 1996; ISBN 0-13-443698-9
- [6] S. J. NORTON, M. D. DIPASQUALE: *Thread Time - The Multithreaded Programming Guide*, Prentice Hall, 1997; ISBN 0-13-190067-6
- [7] D. R. BUTENHOF: *Programming with POSIX Threads*, Addison Wesley, 1997; ISBN 0-201-63392-2
- [8] S. PRASAD: *Multithreading Programming Techniques*, McGraw-Hill, 1996; ISBN 0-079-12250-7
- [9] S. KLEINMAN, B. SMALDERS, D. SHAH: *Programming with Threads*, Prentice Hall, 1995; ISBN 0-131-72389-8
- [10] C.J. NORTHRUP: *Programming With Unix Threads*, John Wiley & Sons, 1996; ISBN 0-471-13751-0
- [11] P. BARTON-DAVIS, D. MCNAMEE, R. VASWANI, E. LAZOWSKA: *Adding Scheduler Activations to Mach 3.0*, University of Washington, 1992; Technical Report 92-08-03
- [12] D. STEIN, D. SHAH: *Implementing Lightweight Threads*, SunSoft Inc., 1992 (published at USENIX'92).
- [13] W.R.STEVENS: *Advanced Programming in the Unix Environment*, Addison-Wesley, 1992; ISBN 0-201-56317-7
- [14] D. LEWINE: *POSIX Programmer's Guide: Writing Portable Unix Programs*, O'Reilly & Associates, Inc., 1994; ISBN 0-937175-73-0
- [15] BRYAN O'SULLIVAN: *Frequently asked questions for comp.os.research*, 1995; <http://www.serpentine.com/~bos/os-faq/>, <ftp://rtfm.mit.edu/pub/usenet/comp.os.research/>
- [16] SUN MICROSYSTEMS, INC: *Threads Frequently Asked Questions*, 1995, <http://www.sun.com/workshop-threads/faq.html>
- [17] BRYAN O'SULLIVAN: *Frequently asked questions for comp.programming.threads*, 1997; <http://www.serpentine.com/~bos/threads-faq/>
- [18] BIL LEWIS: *Frequently asked questions for comp.programming.threads*, 1999; <http://www.lambdacs.com/newsgroup/FAQ.html>
- [19] NUMERIC QUEST INC: *Multithreading - Definitions and Guidelines*; 1998; <http://www.numericquest.com/lang/multi-frame.html>
- [20] THE OPEN GROUP: *The Single Unix Specification, Version 2 - Threads*; 1997; <http://www.opengroup.org/onlinepubs/007908799/xsh/threads.html>
- [21] SUN MICROSYSTEMS INC: *SMI Thread Resources*; <http://www.sun.com/workshop/threads>
- [22] FRANK MUELLER: *FSU pthreads*; 1997; <http://www.cs.fsu.edu/~mueller/pthreads/>
- [23] CHRIS PROVENZANO: *MIT pthreads*; 1993; <http://www.mit.edu/people/proven/pthreads.html> (old), <http://www.humanfactor.com/pthreads/mit-pthreads.html> (updated)
- [24] KOTA ABE: *Portable Threading Library (PTL)*; 1999; <http://www.media.osaka-cu.ac.jp/~k-abe/PTL/>
- [25] RALF S. ENGELSCHALL: *GNU Portable Threads (Pth)*; 1999; <http://www.gnu.org/software/pth/>, <ftp://ftp.gnu.org/gnu/pth/>
- [26] MICHAEL T. PETERSON: *POSIX and DCE Threads For Linux (PCThreads)*; 1995; <http://members.a-net/~mtp/PCthreads.html>
- [27] ROBERT S. THAU: *Really Simple Threads (rsthreads)*; 1996; <ftp://ftp.ai.mit.edu/pub/rst/>
- [28] JOHN BIRRELL: *FreeBSD uthreads*; 1998; [ftp://ftp.freebsd.org/pub/FreeBSD/FreeBSD-current/src/lib/libc\\_r/uthread/](ftp://ftp.freebsd.org/pub/FreeBSD/FreeBSD-current/src/lib/libc_r/uthread/)
- [29] XAVIER LEROY: *The LinuxThreads library*; 1999; <http://pauillac.inria.fr/~xleroy/linuxthreads/>
- [30] IBM: *AIX Version 4.3 General Programming Concepts: Writing and Debugging Programs; Understanding Threads*; 1998; [http://www.rs6000.ibm.com/doc.link/en\\_US/a\\_doc.lib/aixprgdc/genprogdc/understanding-threads.htm](http://www.rs6000.ibm.com/doc.link/en_US/a_doc.lib/aixprgdc/genprogdc/understanding-threads.htm)
- [31] *Netscape Portable Runtime (NSPR)*; <http://www.mozilla.org/docs/refList/refNSPR/>, <http://lxr.mozilla.org/seamonkey/source/nsprpub/>
- [32] RALF S. ENGELSCHALL: *Simple Machine Context Benchmark*; 2000; <http://www.gnu.org/software/pth-/smcb.tar.gz>



# Transparent Run-Time Defense Against Stack Smashing Attacks

Arash Baratloo and Navjot Singh  
{arash,singh}@research.bell-labs.com  
Bell Labs Research, Lucent Technologies  
600 Mountain Ave  
Murray Hill, NJ 07974 USA

Timothy Tsai\*  
ttsai@rstcorp.com  
Reliable Software Technologies  
21351 Ridgetop Circle, Suite 400  
Dulles, VA 20166 USA

## Abstract

The exploitation of buffer overflow vulnerabilities in process stacks constitutes a significant portion of security attacks. We present two new methods to detect and handle such attacks. In contrast to previous work, the new methods work with any existing pre-compiled executable and can be used transparently per-process as well as on a system-wide basis. The first method intercepts all calls to library functions known to be vulnerable. A substitute version of the corresponding function implements the original functionality, but in a manner that ensures that any buffer overflows are contained within the current stack frame. The second method uses binary modification of the process memory to force verification of critical elements of stacks before use. We have implemented both methods on Linux as dynamically loadable libraries and shown that both libraries detect several known attacks. The performance overhead of these libraries range from negligible to 15%.

## 1 Introduction

As the Internet has grown, the opportunities for attempts to access remote systems improperly have increased. Several security attacks, such as the 1988 Internet Worm [7, 18, 19], have even become entrenched in Internet history. Some attacks merely annoy or occupy system resources. However, other attacks are more insidious because they seize root privileges and modify, corrupt, or steal data.

\*This work was performed while the author was with Lucent Technologies, Bell Labs, Murray Hill, NJ USA.

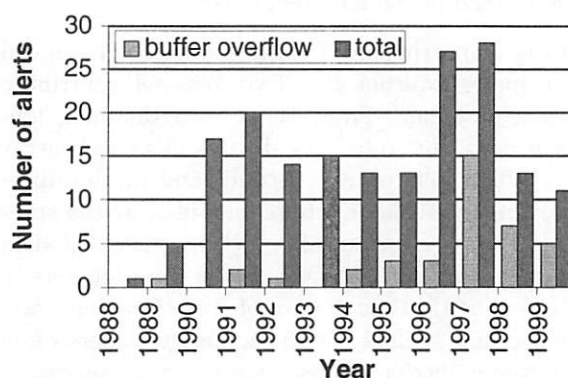


Figure 1: Number of Reported CERT Security Advisories and the Number Attributable to Buffer Overflow (Data from [24])

Figure 1 shows the increase in the number of reported CERT [3] security advisories that are based on buffer overflow. In recent years, attacks that exploit buffer overflow bugs have accounted for approximately half of all reported CERT advisories. The buffer overflow bug may be due to errors in specifying function prototypes or in implementing functions. In either case, an inordinately large amount of data is written to the buffer, thus overflowing it and overwriting the memory immediately following the end of the buffer. The overflow injects additional code into an unsuspecting process and then hijacks control of that process to execute the injected code. The hijacking of control is usually accomplished by overwriting return addresses on the process stack or by overwriting function pointers in the process memory. In either case, an instruction that alters the control flow (such as a call, return, or

Function prototype	Potential problem
<code>strcpy(char *dest, const char *src)</code>	May overflow the <code>dest</code> buffer.
<code>strcat(char *dest, const char *src)</code>	May overflow the <code>dest</code> buffer.
<code>getwd(char *buf)</code>	May overflow the <code>buf</code> buffer.
<code>gets(char *s)</code>	May overflow the <code>s</code> buffer.
<code>fscanf(FILE *stream, const char *format, ...)</code>	May overflow its arguments.
<code>scanf(const char *format, ...)</code>	May overflow its arguments.
<code>realpath(char *path, char resolved_path[])</code>	May overflow the <code>path</code> buffer.
<code>sprintf(char *str, const char *format, ...)</code>	May overflow the <code>str</code> buffer.

Table 1: Partial List of Unsafe Functions in the Standard C Library

jump instruction) may inadvertently transfer execution to the wrong address that points at the injected code instead of the intended code.

Programs written in C have always been plagued with buffer overflows. Two reasons contribute to this problem. First, the C programming language does not automatically bounds-check array and pointer references. Second, and more importantly, many of the functions provided by the standard C library are unsafe, such as those listed in Table 1. Therefore, it is up to the programmers to check explicitly that the use of these functions cannot overflow buffers. However, programmers often omit these checks. Consequently, many programs are plagued with buffer overflows and are therefore vulnerable to security attacks.

Preventing buffer overflows is clearly desirable. If one did not have access to a C program's source code, the general problem of automatically bounds-checking array and pointer references is very difficult, if not impossible. So at first, it might seem natural to dismiss any attempts to perform automatic bounds checking at runtime when one does not have access to the source code. One of the contributions of this paper is to demonstrate that by leveraging some information that is available only at runtime, together with context-specific security knowledge, one can automatically foil security attacks that exploit unsafe functions to overflow stack buffers.

## 2 Buffer Overflow Exploit

The most general form of security attack achieves two goals:

1. Inject the attack code, which is typically a small sequence of instructions that spawns a shell, into a running process.
2. Change the execution path of the running process to execute the attack code.

It is important to note that these two goals are mutually dependent on each other: injecting attack code without the ability to execute it is not necessarily a security vulnerability.

By far, the most popular form of buffer overflow exploitation is to attack buffers on the stack, referred to as the *stack smashing attack*. As is discussed below, the reason for this popularity is because overflowing stack buffers can achieve *both goals simultaneously*. Another form of buffer overflow attack known as the *heap smashing attack*, is to attack buffers residing on the heap (a similar attack involves buffers residing in data space). Heap smashing attacks are much harder to exploit, simply because it is difficult to change the execution path of a running process by overflowing heap buffers. For this reason, heap smashing attacks are far less prevalent.

A complete C program to demonstrate the stack smashing attack is shown in Figure 2. Figure 3 illustrates the address space of a process undergoing this attack. The process stack after executing the initialization code and entering the `main()` function (but before executing any of the instructions) is illustrated in Figure 3(a). Notice the structure of the top stack frame (i.e., the stack frame for `main()`). This stack frame contains, in order, the function parameters, the return address of the calling function, the previous frame pointer, and finally the stack variable `buffer`. Looking at the sample program in Figure 2, a sequence of instructions for spawning a shell is stored in a string variable called `shellcode` (lines 3-6). The `shellcode` is equivalent to execut-



---

```

#include <stdio.h>

char shellcode[] =
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd" 5
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];
int i;
long *long_ptr;

int main() {
    char buffer[96];

    long_ptr = (long *)large_string;
    for (i=0; i<32; i++)
        *(long_ptr+i) = (int)buffer;
    for (i=0; i<(int)strlen(shellcode); i++)
        large_string[i] = shellcode[i];
    strcpy(buffer, large_string);
    return 0;
}

```

---

Figure 2: A Sample Program to Demonstrate a Stack Smashing Attack

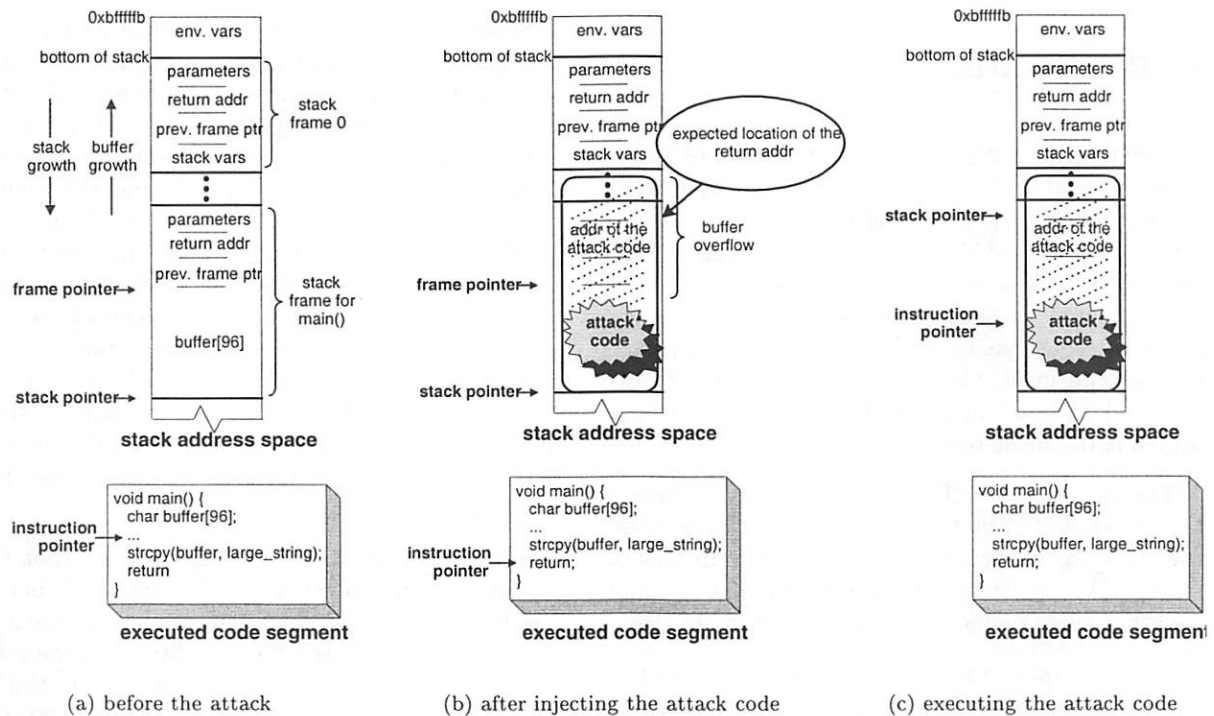


Figure 3: A Process Undergoing a Stack Smashing Attack

ing `exec("/bin/sh")`. The two `for` loops in the `main` function prepare the attack code by writing two sequences of bytes to `large_string`: the `for` loop starting on line 16 writes the (future) starting address of the attack code; then the `for` loop starting on line 18 copies the attack code (excluding the terminating null character). The stack is smashed on line 20 by the `strcpy()` function. Figure 3(b) depicts the process' stack space after executing the `strcpy()` call. Notice how the unsafe use of `strcpy()` simultaneously achieves both requirements of the stack smashing attack: (1) it injects the attack code by writing it on the process' stack space, and (2) by overwriting the return address with the address of the attack code, it instruments the stack to alter the execution path. The attack completes once the `return` statement on line 21 is executed: the instruction pointer "jumps" and starts executing the attack code. This step is illustrated in Figure 3(c).

In a real security attack, the attack code would normally come from an environment variable, user input, or even worse, from a network connection. A successful attack on a privileged process would give the attacker an interactive shell with the user-ID of root, referred to as a root shell.

### 3 Related Work

The Internet Worm that infected tens of thousands of hosts in 1988 was one of the first well-known buffer overflow attacks, although there are some anecdotal evidence that buffer overflow attacks date back to the 1960's [4]. The proportion of attacks based on buffer overflows is increasing each year—in recent years, buffer overflow attacks have become the most widely used type of security attack [24]. Among such attacks, the stack smashing attack is the most popular form [10, 22].

The majority of buffer overflow attacks, including the one exploited by the Internet Worm is based on the stack smashing attack. Detailed descriptions of stack smashing attacks are presented in [20, 22], and cook-book-like recipes are presented in [6, 15, 16].

Researchers in the areas of operating systems, static code analyzers and compilers, and run-time middleware systems have proposed solutions to circumvent stack smashing type of attacks. In most operating systems the stack region is marked as executable, which means that code located in the stack

memory can be executed. Because this "feature" is used by stack smashing attacks, making the stack non-executable is a commonly proposed method for thwarting overflow attacks. A kernel patch removing the stack execution permission has been made available [17]. This approach, however, has some drawbacks. First, patching and recompiling the kernel is not feasible for everyone. Second, *nested function calls* or *trampoline functions*, which are used extensively by LISP interpreters and Objective C compilers, and the most common implementation of signal handler returns on Unix (as well as Linux), rely on an executable stack to work properly. And finally, an alternative attack on stacks known as *return-into-libc*, which directs the program control into code located in shared libraries, cannot be defeated by making the stack non-executable [25]. Because of those reasons, Linus Torvalds has consistently refused to incorporate this change into the Linux kernel [23].

Snarskii has developed a custom implementation of the standard C library for FreeBSD [21]. This library targets the set of unsafe functions, and inspects the process stack to detect buffer overflows that write across frame pointers. In contrast to our work, this is a custom implementation and replaces the standard C library.

Several commonly used tools, such as Lint [11], and those proposed in [8] use compile-time analysis to detect common programming errors. Existing compilers have also been augmented to perform bounds-checking [13]. These projects have demonstrated limited success in preventing the general buffer overflow problem. Wagner *et al.* have recently proposed the use of compile-time range analysis to ensure the "safe" use of C library functions [24]. This project specifically concentrates on the set of unsafe library functions. Unlike our approach, this method requires access to a program's source code, which is not always available. Moreover, preliminary results indicate that this method may produce false positives: a correct program may produce warning or error messages.

StackGuard [5] is another compiler extension that instruments the generated code with stack-bounds checks. Specifically, on function entry, a *canary* is placed near the caller's return address on the stack. Before the function returns to the caller, the validity of this canary is checked and the program is terminated if a discrepancy is detected. This approach works on the assumption that if the return address is tampered with (due to buffer overflows),

Program Name	Version	Description	Result of Attack	Result with libsafe or libverify
xlockmore	3.10	Lock an X Window display	root shell	terminated
amd	6.0	Automatic remote file system mount daemon	root shell	terminated
imapd	3.6	IMAP mail server	root shell	terminated
elm	2.5 PL0pre8	ELM mail user agent	root shell	terminated
SuperProbe	2.11	Probes and identifies video hardware	root shell	terminated

Table 2: List of Some Known Exploits That Are Detected

the canary will also be modified, thus causing validation of the canary to fail. With the exception of a few programs, this approach has shown to be effective. StackGuard introduces a noticeable run-time overhead. Furthermore, StackGuard requires source code access, and there are some programs, such as Netscape Navigator, Adobe Acrobat Reader, and Star Office, that it does not currently support.

Janus [9] is a run-time sand-boxing environment that confines each application to a set of predefined operations. It works on the principle that “an application can do little harm if its access to the underlying operating system is appropriately restricted.” It relies on the operating system’s debugging features, such as `trace` and `strace`, to observe and to confine a process to a sand-box. Similar to our work, this approach works with existing binary applications and does not require access to application’s source code. However, unlike our approach, Janus does not work with applications that legitimately need high privileges. For example, the Unix `login` process requires a high level of privilege to execute, but Janus is unable to selectively allow legitimate privileges while denying unauthorized privileges. This inherent limitation prevents Janus from being applied to high privileged applications, where secure execution is most critical.

## 4 Overview of Techniques

This paper presents two novel methods for performing detection and handling of buffer overflow attacks. In contrast to previous methods and without requiring access to a program’s source code, our novel methods can transparently protect processes against stack smashing attacks, even on a system-wide basis. The first method intercepts all calls to library functions that are known to be vulnerable.

A substitute version of the corresponding function implements the original functionality, but in a manner that ensures that any buffer overflows are contained within the current stack frame. This method has been implemented as a dynamically loadable library called *libsafe*. The second method uses binary re-writing of the process memory to force verification of critical elements of stacks before use. This method has also been implemented as a dynamically loadable library called *libverify*.

The key idea behind *libsafe* is the ability to estimate a safe upper limit on the size of buffers automatically. This estimation cannot be performed at compile time because the size of the buffer may not be known at that time. Thus, the calculation of the buffer size must be made after the start of the function in which the buffer is accessed. Our method is able to determine the maximum buffer size by realizing that such local buffers cannot extend beyond the end of the current stack frame. This realization allows the substitute version of the function to limit buffer writes within the estimated buffer size. Thus, the return address from that function, which is located on the stack, cannot be overwritten, and control of the process cannot be commandeered.

The *libverify* library relies on verification of a function’s return address before use, a scheme similar to that found in StackGuard. The difference is the manner of implementation. Whereas StackGuard introduces the verification code during compilation, *libverify* injects the verification code at the start of the process execution via a binary re-write of the process memory. Furthermore, *libverify* uses the actual return address for verification instead of a “canary” value representing the return address. Thus, in contrast to StackGuard, *libverify* can protect pre-compiled executables.

We have implemented the previously described methods as dynamically loadable libraries on Linux

	Instrumentation Techniques					
	None	libsafe	libverify	StackGuard	Janus	Non-Executable Stack
Effectiveness (what types of errors are handled?)						
Kernel Errors	No	No	Yes	Yes	No	Yes
Specification Errors	No	Yes	Yes <sup>a</sup>	Yes <sup>a</sup>	Maybe <sup>b</sup>	Maybe <sup>c</sup>
Implementation Errors	No	Maybe <sup>d</sup>	Yes <sup>a</sup>	Yes <sup>a</sup>	Maybe <sup>b</sup>	Maybe <sup>c</sup>
User Code Errors	No	No	Yes	Yes	Maybe <sup>b</sup>	Maybe <sup>c</sup>
Other characteristics						
Performance Overhead	None	Very low	Medium	Medium	Medium	None
Disk Usage Overhead	None	Very low	Very low	Low	Very low	None
Source Code Needed	No	No	No	Yes	No	No
Ease of Use	—	Very easy	Very easy	Medium <sup>e</sup>	Easy-Medium <sup>f</sup>	Easy-Medium <sup>g</sup>

<sup>a</sup>If libraries are instrumented.

<sup>b</sup>Cannot catch hijacked privileges that are similar to legitimate privileges.

<sup>c</sup>For certain types of exploits (see Section 3).

<sup>d</sup>If we know which functions have errors.

<sup>e</sup>Source code must be recompiled, and the compiler may also need to be recompiled.

<sup>f</sup>Policies need to be written.

<sup>g</sup>Kernel may need to be patched and recompiled.

Table 3: Summary of Detection Technique Characteristics

and tested them against several security attacks. Table 2 lists several commonly used applications and the result of running publicly available exploits against the applications with and without our libraries.<sup>1</sup> As the table indicates, libsafe and libverify were able to detect the exploits and terminate the programs before any serious harm was done.

The characteristics of libsafe and libverify are shown in Table 3 along with the corresponding characteristics of alternative methods: StackGuard, Janus, and kernel patches for non-executable stack, which were described earlier in Section 3. The first instrumentation technique labeled “None” is presented as a point of comparison and represents the original program with no modifications. The upper half of Table 3 describes the types of errors that each method is able to handle. Specification and implementation errors refer to errors in standard library functions. In particular, by specification errors we mean the set of functions known to be unsafe as described in Section 1; implementation errors refer to the set of functions that are unsafe due to implementation errors. Kernel errors and user code errors

refer to implementation errors in kernel code and user code, respectively. The bottom half of the table describes other characteristics. The performance overhead includes only the run-time overhead. Time spent during configuration and compilation are not included. The disk usage overhead is the extra disk space required due to additional shared libraries, increased executable binary file sizes, and configuration files. The next to last row indicates whether access to source code of the defective program is needed. The ease of use considers the complexity and time requirement of human efforts needed for configuration and compilation.

## 5 Libsafe

The fundamental observations forming the basis of the libsafe library are the following:

- Overflowing a stack variable—that is, injecting the attack code into a running process—does not necessarily lead to a successful stack smashing attack. The attack must also divert

<sup>1</sup>The security attacks are available from Crv’s Security Bugware Page (<http://oliver.efri.hr/~crv/>).



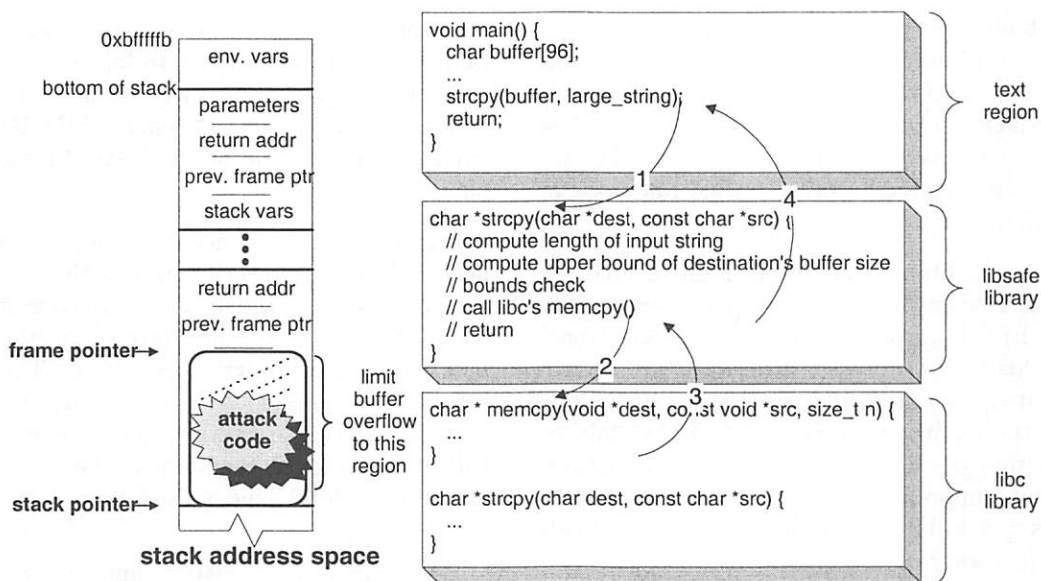


Figure 4: Libsafe Containment of Buffer Overflow

the execution sequence of a process to run the attack code.

- Although buffer overflows cannot be stopped in general, automatic and transparent run-time mechanisms can prevent the overflow from corrupting a return address and altering the control flow of a process.

Refer to Figure 3(a) for an example. At the time `strcpy()` is called, the frame pointer (i.e., the `ebp` register in the Intel Architecture) will be pointing to a memory location containing the previous frame's frame pointer. Furthermore, the frame pointer separates the stack variables (local to the current function) from the parameters passed to the function. Continuing with the example of Figure 3(a), the size of `buffer` and all other stack variables residing on the top frame cannot extend beyond the frame pointer—this is a safe upper limit. A correct C program should never explicitly modify any stored frame pointers, nor should it explicitly modify any return addresses (located next to the frame pointers). We use this knowledge to detect and limit stack buffer overflows. As a result, the attack executed by calling the `strcpy()` can be detected and terminated before the return address is corrupted (as in Figure 3(b)). In the case that a local buffer on one of the previous stack frames is accessed, then frame pointers are traversed up the stack until the right stack frame is found, and then libsafe computes the upper bound.

Libsafe implements the above technique. It is implemented as a dynamically loadable library that is preloaded with every process it needs to protect. The preloading injects the libsafe library between the program code and the dynamically loadable standard C library functions. The library can then intercept and bounds-check the arguments before allowing the standard C library functions to execute. In particular, it intercepts the unsafe functions listed in Table 1 to provide the following guarantees:

- Correct programs will execute correctly, i.e., no false positives.
- The frame pointers, and more importantly return addresses, can never be overwritten by an intercepted function—an overflow that would lead to overwriting the return address is always detected.

Figure 4 illustrates the memory of a process that has been linked with the libsafe library, and in particular, it shows the new implementation of `strcpy()` in the libsafe library. Once the program invokes `strcpy()`, the version implemented in the libsafe library gets executed—this is due to the order in which the libraries were loaded. The libsafe implementation of the `strcpy()` function first computes the length of the source string and the upper bound on the size of the destination buffer (as explained above). It then verifies that the length of the source string is less than the bound on the des-

termination buffer. If the verification succeeds, then the `strcpy()` calls `memcpy()` (implemented in the standard C library) to perform the operation. However, if the verification fails, `strcpy()` creates a `syslog` entry and terminates the program. A similar approach is applied to the other unsafe functions in the standard C library.

The `libsafe` library has been implemented on Linux. It uses the preload feature of dynamically loadable ELF libraries to automatically and transparently load with processes it needs to protect. In essence, it can be used in one of two ways: (1) by defining the environment variable `LD_PRELOAD`, or (2) by listing the library in `/etc/ld.so.preload`. The former approach allows per-process control, whereas the latter approach automatically loads the `libsafe` library machine-wide.

The `libsafe` library does not use any Linux specific feature of ELF; these ELF features are available for many other versions of Unix such as Solaris, and have been used for other purposes [1, 14]. Furthermore, an alternative technique with a similar feature can be used for Windows NT [2, 12].

We have installed the `libsafe` library on a Linux machine. The library is automatically loaded with every process and transparently protects each process from stack smashing attacks. The protected applications include daemon processes such as the Apache HTTP server, `sendmail`, and an NFS server, as well as those started by users such as the XFree86 server, the Enlightenment window manager, GNU Emacs, Netscape Navigator, and Adobe Acrobat Reader. We have used this machine for several months and found the machine to be stable and running without a noticeable performance hit.

## 6 Libverify

The `libverify` library implements a return address verification scheme similar to that used in StackGuard.

Both methods protect return addresses on the process stack by saving canary values at the start of a function and verifying the canary value at the end of the function to determine if any buffer overflow occurred. However, in contrast to StackGuard, `libverify` requires no recompilation of source code and is therefore applicable to legacy programs. Instead, all code for saving and verifying canaries is

contained in a special library. This library also contains instrumentation code to link the canary code with the program. As with `libsafe`, the library is activated by specifying it as part of the `LD_PRELOAD` environment variable or the `/etc/ld.so.preload` file.

Figure 5 shows the memory of a process that has been linked with `libverify`. Before the process commences execution, the library is linked with the user code. As part of the link procedure, the `_init()` function in the library is executed. The `_init()` function contains code to instrument the process such that the canary verification code in the library will be called for all functions in the user code. The instrumentation includes the following steps:

1. Determine the location and size of the user code.
2. Determine the starting addresses of all functions in the user code.
3. For each function
  - (a) Copy the function to heap memory.
  - (b) Overwrite the first instruction of the original function with a jump to the `wrapper_entry` function.
  - (c) Overwrite the return instruction of the copied function with a jump to the `wrapper_exit` function.

The `wrapper_entry` function saves a copy of the canary value on a canary stack and then jumps to the copied function. The `wrapper_exit` function verifies the current canary value with the canary stack. A canary stack is needed to save canary values for nested function calls. If the canary value is not found on the canary stack, then the function determines that a buffer overflow has occurred. In that case, the `wrapper_exit` function then calls the `die()` function, which creates a `syslog` entry, prints an error message to the standard error device, and terminates. The `die()` function can also perform additional notification and handling, such as sending an email message or shutting down the entire system.

In contrast to StackGuard, which generates random numbers for use as canaries, `libverify` uses the actual return address as the canary value for each function. This simplifies the binary instrumentation procedure because no additional data is pushed onto the stack, which means that the relative offsets to all data within each stack frame remain the same. Although the return address can sometimes be guessed

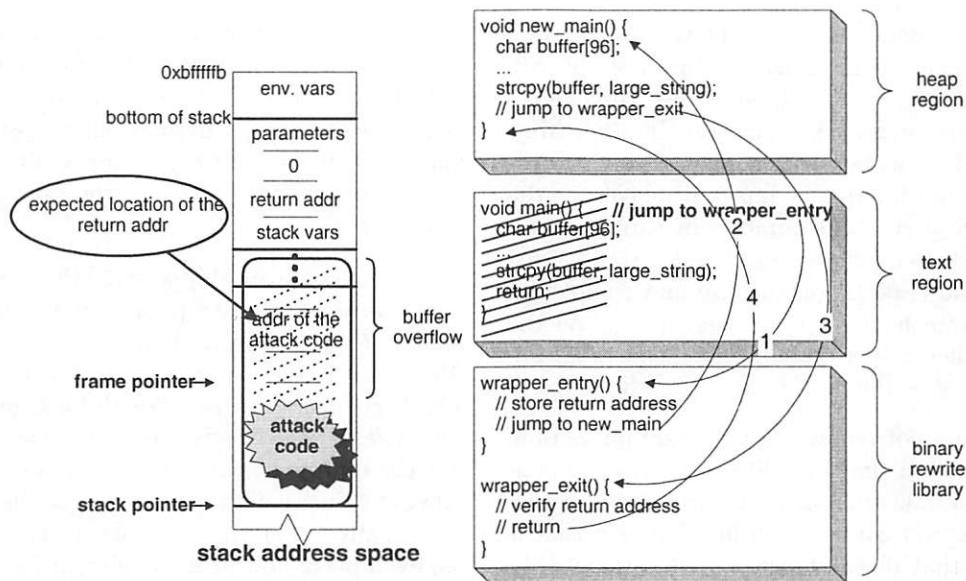


Figure 5: Memory Usage for libverify

by an attacker, control flow is still protected because the actual value of any return address is explicitly verified before execution of that return instruction. The canary stack resides in heap memory. The size is dynamically extended to accommodate a large number of simultaneous canaries. The canary stack itself is not protected against overflow attacks in the current libverify implementation. However, such protection can be easily added by using the `mprotect()` function to designate the page immediately preceding the canary stack as non-writable.

A difficulty does arise when a function performs an absolute jump to an address within the same function. As an example, this situation might occur for some `switch()` statements. Because we copy the original function to heap memory and execute that function from the copied version, an absolute jump in the copied function would force control flow to the original function. To handle this situation, we overwrite the original function with trap instructions. If control is forced to the original function, the trap is activated, and a trap handler returns control flow back to the copied function.

## 7 Experiments

The libsafe and libverify libraries are effective in detecting and defeating stack smashing attacks. Extra code is needed to perform this detection, and that extra code incurs a performance overhead. In

Application	Size (Bytes)	Initialization time ( $\mu$ s)
quicksort	27330	13032
imapd	1305379	67491
tar	418283	40334
xv	1242686	195205

Table 4: The Initialization Elapsed Times for libverify Library

this section we quantify the performance overhead associated with use of these libraries. Section 7.1 describes the overheads associated with micro benchmarks to illustrate the range of possible overheads. Section 7.2 gives performance data for a selected set of actual applications.

All experiments were conducted on a 400 MHz Pentium II machine with 128 MB of memory running RedHat Linux version 6.0. Our libraries and all programs in Sections 7.1 and 7.2 were compiled (and optimized using `-O2`) with GCC compiler version 2.91.66.

### 7.1 Micro Benchmarks

As the part of the link procedure, libverify executes its initialization section, the `_init()` function), as described in Section 6. This initialization section first reads, then copies and modifies

the entire instruction sequence of the application. Table 4 presents the initialization times of libverify with four commonly used applications: `quicksort` (a fast sorting program), `imapd` (an Internet Message Access Protocol server), `tar` (an archiving utility), and `xv` (an interactive image displayer for the X Window System). The numbers in Table 4 represent the start-up overhead associated with libverify. This overhead depends on the size and complexity of the program libverify is instrumenting. As the numbers indicate, the start-up overhead takes approximately 50 – 160 milliseconds per Megabyte.

Libsafe does not require an initialization section. However, the first time each libsafe function is activated, the initialization of that particular function makes a `dlsym()` call for each libc function that is called from that libsafe function. Because the libsafe function has the same name as the corresponding libc version, the `dlsym()` call is needed to obtain a pointer to the libc function. Each `dlsym()` call requires 1.26  $\mu$ s. The interception and redirection of a C library function consists of an additional user-level function call, which approximately adds 0.04  $\mu$ s of overhead.

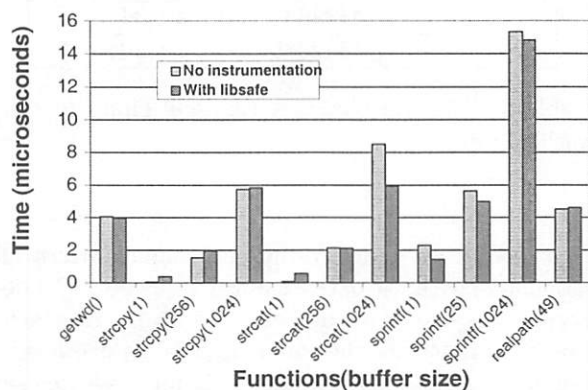


Figure 6: Performance of Libsafe Functions

To quantify the performance overhead of the libsafe library we measured the execution times of five unsafe C library functions and compared the results with our “safe” versions. The results are depicted in Figure 6. Reported times are “wall clock” elapsed times as reported by `gettimeofday()`. An interesting observation is that the libsafe versions of several functions outperform the original versions. This is a repeatable behavior, and we have observed consistent findings on different machines and operating system versions. This effect is due both to low-level optimizations and the fact that libsafe’s implementation of most functions is different than those of

C library. For example, consider the performance of the `getwd()` and `sprintf()` functions. Our libsafe library replaces these functions with equivalent safe versions. In particular, `getwd()` is replaced with `getcwd()` and `sprintf()` is replaced with `snprintf()`; on Linux, the safe versions execute faster.

The figure also shows that the libsafe library can slow down the string operations `strcpy()` and `strcat()` by as much as 0.5  $\mu$ s per function call. However, as the string size increases, the absolute overhead decreases because the execution time of the safe versions increases more slowly than that for the unsafe versions. In fact, the safe version of `strcat()` used with strings longer than 256 bytes is actually faster than the unsafe version! This is an example of how using a different implementation (e.g., using `memcpy()` to copy a string) can outperform the standard implementation for certain cases.

The slowdown effect of `strcpy()` is observed in the `realpath()` experiment. When a program calls `realpath()`, the libsafe library calls `realpath()` but stores the result in a buffer in its own memory region. It then uses `strcpy()` to copy the result to the final destination. As Figure 6 shows the slowdown effect of `strcpy()` on `realpath()` is less than 0.05  $\mu$ s.

## 7.2 Application Benchmarks

Since we propose that the libraries are best used on a machine-wide bases to protect against yet unknown attacks, their performance impact is important for all commonly used application. We used four real-world applications to illustrate the performance overhead of our libraries. The applications are `quicksort` (a CPU-bound program) ordering 1,000,000 integers, `imapd` (a network-bound program) transmitting 100 email messages of size 2 kilobyte each, `tar` (an I/O-bound program) archiving 5 Megabytes of data, and `xv` (a CPU and video-bound program) displaying a 1.2 Megabyte image. Figure 7 shows the execution time for each of these applications (1) unmodified and without any security measure, (2) using the libsafe library, (3) using the libverify library, and (4) compiled with StackGuard.

The execution times are based on 100 runs and are given in seconds, with associated 95% confidence intervals. Reported times are elapsed times as reported by `/bin/time`, and include the extra initial-



ization time required by libverify.

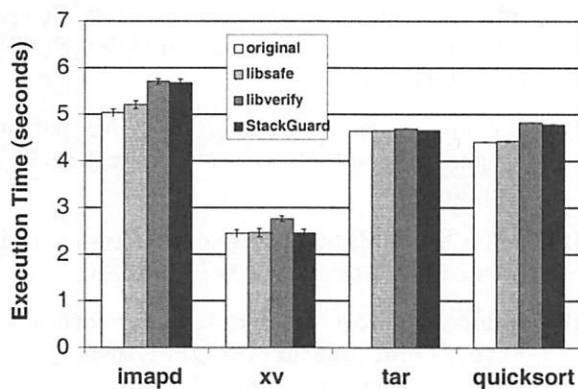


Figure 7: Mean Execution Times (With 95% confidence intervals) of Sample Applications

Figure 7 shows that the overheads associated with all detection methods are reasonable (i.e., less than 15% for these applications). Libsafe is the most efficient method because only the unsafe library functions are intercepted.

Libverify incurs a greater overhead than the libsafe because all user functions are verified. For most of the applications, the overhead is similar to that for StackGuard because the same number of functions is verified. For xv, the need to handle a large number of traps (as described in Section 6) increases the overhead. The overall application test results are encouraging, particularly with libsafe. We have installed and used libsafe on one of our own machines, and have found that the overhead is not noticeable in practice.

## 8 Conclusions

We have described two complementary methods for foiling stack smashing attacks that rely on corrupting the return address, and implemented these methods as dynamically loaded libraries called libsafe and libverify.

An interesting finding is the performance of libsafe. We anticipated a low performance overhead at the onset of the project. We were happily surprised to find how little this overhead is in practice. Because of low-level optimizations and because libsafe's implementation of most functions is different than those of C library, for some applications we actually observed a speedup. This is encouraging since

it indicates the viability of this approach. Furthermore, the elegance and simplicity of instrumenting the standard C library led to a stable implementation.

The implementation of libverify gave us quite a challenge. Our initial goal in re-writing binary instruction streams was to insert the minimum amount of code at beginning of each function to divert the execution control to the `wrapper_entry`, and similarly, to insert the minimal code at the end of the function to execute `wrapper_exit` before returning to the caller. However on the Intel Architecture, we could not fit the required instructions at the end of each function. Hence, we settled with copying the entire function to the heap where space was not a limitation. Relocating functions from the text region to the heap gave rise to the problems we encountered with absolute jumps (as discussed in Section 6). Furthermore, it doubled the code space required for each process. We believe this approach to verifying return addresses is well suited for RISC architectures such as the Alpha or SPARC where the instructions are all the same size.

We believe that the stability, minimal performance overhead, and ease of use (i.e., no modification or recompilation of source code) of the two libraries makes them an attractive first line of defense against stack smashing attacks. It is generally accepted that the best solution to buffer overflow attacks is to fix the original defects in the programs. However, fixing the defects requires knowing that a particular program is defective. The true benefit of using libsafe and libverify is protection against attacks on programs that are not yet known to be vulnerable.

## 9 Acknowledgments

We are thankful to Scott Alexander for his insightful comments and to Vandoorselaere Yoann for his assistance with the software.

## 10 Availability

The libsafe library is available under the GNU Library General Public License. Further information is available from <http://www.bell-labs.com/org/11356/libsafe.html>.

## References

- [1] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. Extending the operating system at the user-level: the Ufo global file system. In *Proceedings of the 7th USENIX Annual Technical Conference*, 1997.
- [2] Robert Balzer and Neil Goldman. Mediating connectors. In *Proceedings the 19th IEEE International Conference on Distributed Computing Systems Workshop*, 1999.
- [3] CERT coordination center. <http://www.cert.org>.
- [4] Crispin Cowan. [http://geek-girl.com/bugtraq/1999\\_1/0481.html](http://geek-girl.com/bugtraq/1999_1/0481.html), 1999. Posting to Bugtraq Mailing List.
- [5] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, 1998.
- [6] dark spyrit aka Barnaby Jack. Win32 buffer overflows (location, exploitation and prevention). <http://www.insecure.org>.
- [7] Mark W. Eichin and Jon A. Rochlis. With microscope and tweezers: An analysis of the internet virus of november 1988. In *Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy (SSP '89)*, 1989.
- [8] David Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1996.
- [9] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th USENIX Security Symposium*, 1996.
- [10] Shawn Instenes. Stack smashing: What to do? ;login: the USENIX Association newsletter, April 1997.
- [11] Stephen C. Johnson. *Lint, a C program checker*. Bell Laboratories, Murray Hill, New Jersey, USA, December 1977. Computer Science Technical Report 65.
- [12] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP)*, December 1993.
- [13] Richard Jones. Bounds checking patches for gcc. <http://web.inter.NL.net/hcc/Haj.Ten.Brugge>.
- [14] Alain Knaff. Zlibc - transparent access to compressed file. <http://zlibc.linux.lu>.
- [15] Mudge. How to write buffer overflows. [http://www.insecure.org/stf/mudge\\_buffer\\_overflow\\_tutorial.html](http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html), 1995.
- [16] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), 1998.
- [17] Openwall Project. Linux kernel patch from the openwall project. <http://www.openwall.com/linux>.
- [18] Jon A. Rochlis and Mark W. Eichin. With microscope and tweezers: The worm from MIT's perspective. *Communications of the ACM*, June 1989.
- [19] Donn Seeley. A tour of the worm. In *Proceedings 1989 Winter USENIX Technical Conference*, January 30 - February 3 1989.
- [20] Nathan Smith. Stack smashing vulnerabilities in the UNIX operating system. <http://millcomm.com/~nate/machines/security/stack-smashing/nate-buffer.%ps>, 1997.
- [21] Alexandre Snarskii. Increasing overall security.... <ftp://ftp.lucky.net/pub/unix/local/libc-letter> and <http://www.lexa.ru:8100/snar/libparanoia>, 1997.
- [22] Evan Thomas. Attack class: Buffer overflows. *Hello World!*, 1999.
- [23] Linus Torvalds. Posting to linux kernel mailing list. <http://www.lwn.net/980806/a/linus-noexec.html>, 1998.
- [24] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings 7th Network and Distributed System Security Symposium*, February 2000.
- [25] Rafel Wojtczuk. Defeating solar designer non-executable stack patch. <http://geek-girl.com/bugtraq>, January 1998.

# Towards Availability Benchmarks: A Case Study of Software RAID Systems

Aaron Brown and David A. Patterson

Computer Science Division, University of California at Berkeley

387 Soda Hall #1776, Berkeley, CA 94720-1776

{abrown,patterson}@cs.berkeley.edu

## Abstract

*Benchmarks have historically played a key role in guiding the progress of computer science systems research and development, but have traditionally neglected the areas of availability, maintainability, and evolutionary growth, areas that have recently become critically important in high-end system design. As a first step in addressing this deficiency, we introduce a general methodology for benchmarking the availability of computer systems. Our methodology uses fault injection to provoke situations where availability may be compromised, leverages existing performance benchmarks for workload generation and data collection, and can produce results in both detail-rich graphical presentations or in distilled numerical summaries. We apply the methodology to measure the availability of the software RAID systems shipped with Linux, Solaris 7 Server, and Windows 2000 Server, and find that the methodology is powerful enough not only to quantify the impact of various failure conditions on the availability of these systems, but also to unearth their design philosophies with respect to transient errors and recovery policy.*

## 1 Introduction

There is a consensus emerging in parts of the systems community that the traditional focus on performance has become misdirected in today's world, a world in which the problems of availability, maintenance, and growth have become at least as important as peak performance, if not more so. One need only open up a recent issue of the *New York Times* or *Wall Street Journal* to see evidence of this fact—the number of stories focusing on recent outages of big e-commerce providers and the major business impact of those outages is staggering; furthermore, several of those outages have been reported as resulting from errors made by systems management staff [19]. Even from a financial standpoint, availability and manageability are important: the management costs for servers providing 24x7 service are typically reported as being several times that of the hardware itself [9][11][12].

The research community is beginning to recognize the importance of focusing on maintainability, availability, and growth as well. The attendees of the 7th HotOS workshop concluded that achieving “No-Futz Computing” (incorporating ideas of manageability, reliability, and availability, amongst others) is one of the most pressing challenges facing systems researchers today [20]. And, in his keynote at the 1999 FCRC conference, John Hennessy argued the same point, insisting that “performance should be less of an emphasis. Instead, other qualities will become crucial: *availability* [...], *maintainability*, [...] and] *scalability*. [...] For servers—if access to services on servers is the killer app—availabil-

ity is *the* key metric” [13]. Furthermore, the traditional “scalability problem,” of creating and efficiently using large massively-parallel systems, is giving way to what we call the “evolutionary growth problem”: constructing large-scale servers that can be incrementally expanded using newer, heterogeneous components.

Benchmarks have historically helped shape computer systems research and development, and we believe it is time for them to do so again. It is time for benchmarks to expand past the space of performance measurement and into the realm of quantifying availability, manageability, and growth; once such benchmarks exist, research into these areas will become significantly more tractable and research progress will naturally follow. Thus, as part of the Berkeley ISTORE project [4], we are investigating techniques for building reproducible, cross-platform “AME” benchmarks for Availability, Maintainability, and Evolutionary Growth, the three challenge areas laid out by Hennessy [13].

In this paper, we present our first steps toward that goal. We have chosen to focus initially on availability, and to begin by developing a general benchmarking methodology for measuring availability. As an initial proof of concept, we have applied this methodology to measure the availability of the software RAID-5 implementations that ship with three popular PC server operating systems: Linux, Solaris 7 Server, and Windows 2000 Server. RAID-5 is just the first example, and we hope our approach will inspire others to benchmark availability of other subsystems.

We chose software RAID as a case study for several reasons. First, software RAID implementations are included with many commercial OS releases (such as the server editions of Solaris 7 and Windows 2000) and with all of the major free UNIX-like operating systems, including Linux, which is being increasingly deployed for Internet service applications. More importantly, RAID has well-defined availability goals, making it an ideal candidate application for benchmarking availability. Also, it is not unusual to find software RAID underlying many Internet service applications that demand 24x7 availability, and thus the availability of the RAID implementation plays an important role in that of the service application itself. Finally, although there is agreement on general features of a RAID-5 system, availability benchmarking can highlight RAID implementation decisions that are important to applications but that are not measured or even mentioned today, for example how a RAID system distinguishes between a disk failure and a temporary glitch.

In studying the availability of the software RAID systems, we found significant differences in implementation philosophy between the various OS implementations. The major differences in philosophy between the systems can be classified along two axes: the first measures the system's paranoia with respect to transient errors, while the second measures the relative priorities placed on preserving application performance versus quickly rebuilding redundancy after a failure. On these axes, the Linux software RAID implementation is paranoid about transients but values application I/O performance more than fast post-failure reconstruction. Solaris falls at the opposite end of both spectrums, demonstrating a near-complete tolerance for transient errors and emphasizing fast reconstruction despite its potential impact on application performance. Windows 2000 falls between Linux and Solaris, although it lies closest to the Solaris end of the spectrum: it tolerates a set of transient errors that is only slightly less robust than Solaris's, and demonstrates a reconstruction philosophy that is similarly aggressive but more workload-aware than Solaris's. The fact that our benchmarks could reveal these philosophies despite treating the implementations as black boxes highlights the power of the methodology.

The remainder of this paper is organized as follows. First, we describe our generic methodology for availability benchmarking in Section 2. In Section 3, we show how that methodology was specialized for the case of measuring software RAID availability, and describe our experimental approach. We present our availability results and describe our experience with the benchmarks and RAID systems in Section 4. Section 5 discusses related work, Section 6 presents our future plans for this work, and we conclude in Section 7.

## 2 A General Methodology for Availability Benchmarking

In this section, we describe a general methodology that can be used to measure and study the availability of arbitrary computer systems. We begin by establishing a standard definition of availability and the metrics that can be used to report it, then consider how to construct benchmarks that produce those metrics, and finally describe how the results of those benchmarks can be reported and analyzed.

### 2.1 Availability: definitions and metrics

The term "availability" carries with it many possible connotations. Traditionally, availability has been defined as a binary metric that describes whether a system is "up" or "down" at a single point of time. A traditional extension of this definition is to compute the percentage of time, on average, that a system is available ("up") or not ("down")—this is how availability is defined when a system is described as having 99.999% availability, for example.

We take a different perspective on availability. First, we see availability as a spectrum, and not a binary metric. Systems can exist in a large number of degraded, but operational, states between "down" and "up." In fact, systems running in degraded states are probably more common than "perfect" systems [2], especially in the fast-growing world of online service provision where economic pressures encourage deployment of less-well-tested commodity SMP- and cluster-based servers rather than expensive fault-tolerant machines. An availability metric must therefore capture these degraded states, measuring not only whether a system is up or down, but also its efficacy, or the quality of service that it is providing.

Second, availability must not be defined at a single point in time or as a simple average over all time. It must instead be examined as a function of the system's quality of service over time. To motivate this, consider that from a user's perspective, there is a big difference between a system that refuses requests for two seconds out of every minute and one that is down for one whole day every month, even though the two systems have approximately the same average uptime. Any benchmark of availability must be able to capture the difference between those two systems.

Combining these two requirements, we propose that availability be measured by examining the variations in system quality of service metrics over time. The particular choice of quality of service metrics depends on the type of system being studied. Two obvious metrics that apply to most server systems are performance and degree of fault-tolerance. For a web server, these metrics would map to requests satisfied per second (or per-



haps latency of request service) and the number of failures that can be tolerated by the storage subsystem, network connection topology, and so forth. Other possible metrics might include:

- *completeness*: consider a system like the Inktomi search engine that tolerates failures by returning search results that cover only the remaining available parts of its database [10];
- *accuracy*: a system that must perform a large computation in a fixed amount of time (e.g., decoding real-time media) might sacrifice accuracy in the computation when running in degraded mode; and
- *capacity*: to maintain other metrics while in a degraded state, a system might limit the number of clients or jobs it will accept, or might discontinue less-essential services.

We discuss how these time-dependent availability measurements might be concretely represented as graphs and numerical summary statistics in Section 2.3, below.

## 2.2 Towards an availability benchmarking methodology

Having selected the availability and quality-of-service metrics for a given type of system, our next challenge is to accurately and reproducibly measure them in a controlled benchmarking environment. Doing so is complicated by the fact that typical benchmark environments are explicitly designed to prevent the kinds of exceptional behavior that would cause availability to be affected in real-world systems.

Thus, in order to perform availability benchmarks, it is necessary to have a benchmark environment that provides a means of generating fault-provoking stimuli and “*maintenance events*” and applying them to the system under test. (A maintenance event is any action taken by a human administrator to maintain, repair, or upgrade the system.) The primary technique that enables such an environment to be constructed is direct *fault injection* into the system under test [1][5]. For example, disk failures in a storage array can be simulated, memory can be artificially corrupted, processes can be killed, power glitches can be simulated, network links can be broken, and so forth. Fault injection need not be limited to hardware faults, however: stimuli such as load spikes, invalid client/user requests, and other workload-driven ways of triggering boundary conditions are also reasonable events to simulate.

To build an availability benchmark, we also need a way to generate a realistic workload and to measure the appropriate quality of service metrics. Our task is simplified by leveraging the extensive efforts at fair workloads from the performance benchmarking community. We simply use existing performance benchmarks to

generate a representative workload for the type of system under test, and to measure the desired metrics at a single point in time. These workload-generating performance benchmarks should be adapted to run continuously, repeatedly measuring the desired metric. The system under test may also need to be modified to measure certain metrics (such as accuracy or completeness).

Given a benchmark environment supporting fault injection and a performance benchmark configured as both a continuous workload generator and a quality of service data collector, running an availability benchmark consists of two steps. First, the workload generator is run without injecting faults and several traces of the values of the desired metrics are recorded. This step establishes a baseline measurement for a non-faulty system. Second, the workload generator is run while simultaneously injecting a *fault workload*, and again a trace of the values of the desired metrics is recorded. This second step is key, since it produces a trace of the behavior of the system’s quality of service over time in response to various faults, which is exactly the time-dependent availability metric that is desired.

The only part of the methodology we have not yet discussed is the content of a “*fault workload*”. As its name suggests, a fault workload is a collection of faults and maintenance events designed to mimic a real-world failure situation.

We see the need for two different kinds of fault workloads, described in the following two sections, roughly corresponding to traditional micro- and macro-benchmarks:

**Single-fault workloads.** The first kind of fault workload is the availability analogue of a performance micro-benchmark. A single-fault workload, as its name implies, consists of just a single fault: once the system under test has reached steady-state, a single fault is injected—such as a disk sector write error—and the system’s behavior (as reflected in the quality of service metrics) is recorded. Intervention of a human administrator in response to the fault is not allowed. Like performance microbenchmarks, single-fault availability benchmarks are most useful for studying isolated pieces of a system and for uncovering design decisions, design flaws, and bugs. Their scope is broader than performance microbenchmarks, however, since a single fault can often have a ripple effect and affect a system as much as a multi-fault workload.

**Multi-fault workloads.** The second kind of fault workload is the availability equivalent of a performance macro-benchmark. Multi-fault workloads consist of a series of faults and maintenance events designed to mimic real-world fault scenarios, for example, a disk failure in a RAID system followed by replacement of the failed

disk followed by a write failure while reconstructing the array. Like traditional application performance macrobenchmarks, multi-fault workloads are useful for building availability benchmarks designed to help select or evaluate new systems, and to identify potential weaknesses in existing systems that need to be addressed. They are also very useful for studying the behavior of the system under pathological failure conditions (as in the RAID example above).

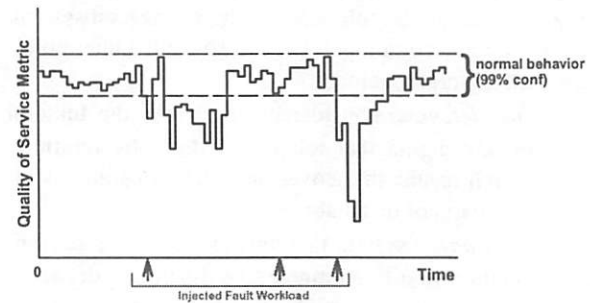
A challenging problem in developing benchmarks based on multi-fault workloads lies in how to realistically and reproducibly simulate the behavior of a human administrator in maintaining the system and in responding to failures originating from fault injection. Such “maintenance events” cannot be ignored, as very few modern systems are truly self-maintaining and most will require human intervention to complete the scenarios. We believe that the solution is to use logs of administrator activity to develop a stochastic model of the system maintenance process and of how administrators typically behave in response to various system failures and stimuli. For example, one might characterize the distribution of response time between a reported disk failure and the replacement of that disk. Such a model can then be used to direct the human intervention in the system during the benchmark run. There is a parallel here to performance benchmarks designed for systems that require human interaction; often in these benchmarks, a script plays back what a person would type in response to prompts. We are currently pursuing this approach.

Note that disk improvements over the years mean that disks no longer fail fast: the classic head crash of operating systems lore almost never happens today, as disks have become physically smaller and their mean time between failures has increased from 50,000 to 1,000,000 hours. Observations of the Tertiary Disk (TD) system at UC Berkeley, a large disk and web server farm, suggest that modern components start acting erratically rather than failing fast, and so a system administrator is much more likely to “fire” and replace an erratic component than to wait for it to fail completely [23]. We feel it is important to capture this type of activity in any model of administrator behavior.

### 2.3 Analyzing and reporting availability benchmark results

The raw data produced from either a single-fault- or multi-fault-workload availability benchmark is rather unwieldy, and therefore some standard techniques for analyzing and reporting it are required.

The simplest way to handle the data from the runs with fault injection is to plot it graphically, with the quality of service metrics on the vertical axis and time on the horizontal axis. The graph is then overlaid with



**Figure 1:** Example availability graph, showing the variation in an application quality of service metric (on the vertical axis) over time (on the horizontal axis), as faults are injected into the system (the faults are represented by heavy arrows). The dashed lines define a 99% confidence interval around the system’s normal (non-faulty) behavior.

confidence intervals calculated from the runs in which no faults were injected; these intervals indicate the range of quality of service values that are statistically “normal.” Finally, the times at which faults were injected are marked on the graph. An example of this type of graph is shown in Figure 1.

These graphs provide a good means by which the experimenter or system designer can study and understand the availability behavior of the system, and they are what we will use later in this paper to report our results for software RAID. In particular, the experimenter can use these graphs to focus on the points at which the measured values of the quality of service metrics fall outside the statistically normal range; these are the points where the system’s availability has been compromised.

However, the graphs remain somewhat difficult to quantify and compare, especially if the benchmarks are to be used by end-users or customers. Several SPEC benchmarks do report graphs, and some customers do compare the graphs side-by-side. But the salient features of the graphs can also be distilled numerically. The most direct approach here is to identify all deviations from the statistically normal range, and to characterize—via mean, standard deviation, and possibly a distribution function—the distributions of the frequency of those deviations, the length of those deviations, and the severity (height) of those deviations. By characterizing the distribution rather than just averaging, this approach may preserve, for example, the distinction between the system that is down 2 seconds every hour and the one that is down one day every month. Of course, these characterizations can be distilled further, for example by simply reporting the product of the average length and average severity of the deviations, although at this point the benchmark result begins to lose much of its descriptive power.

### 3 Implementing the Methodology for Software RAID

In the previous section, we presented a general methodology for benchmarking system availability. In this section, we describe how we implemented that methodology for measuring the availability of the software RAID implementations provided by Linux, Solaris 7 Server, and Windows 2000 Server.

The availability guarantees of RAID-5 are straightforward [7]. A RAID-5 volume can tolerate a single disk failure without loss of data. After that first failure, the volume can continue to service requests in “degraded” mode, although I/Os tend to be more expensive due to the need to reconstruct data on-the-fly. A second disk failure renders the data on the volume inaccessible. Some RAID-5 implementations support spare disks, and can restore redundancy by rebuilding onto the spare after the first failure; during this reconstruction period, the volume will still be destroyed if a non-spare disk fails, although failure of the spare disk can be tolerated.

#### 3.1 Fault injection environment

For the experiments in this paper, we chose to limit the fault injection to faults affecting the disks comprising the software RAID volume, as those are the primary hardware failure points in a software RAID system. Since we wanted to generate a range of different disk faults in a controlled manner, we rejected the simplistic fault-injection technique of pulling disks out of a live system. Instead, we replaced one of the SCSI disks in the software RAID volume with an *emulated disk*, a PC running special software with a special SCSI controller that makes the combination of PC+controller+software appear to other devices on the SCSI bus as a disk drive (*i.e.*, a SCSI target rather than a SCSI controller). Thus our systems under test saw the PC emulating the disk as a real disk drive.

Our emulated disk consisted of an AMD-K6-2-350 PC with an ASC ASC-U2W SCSI adaptor, running Windows NT with the ASC VirtualSCSI Target Mode Emulation library installed [3]. We adapted the library to emulate one or two SCSI disk drives by converting I/O requests to the emulated disk into reads and writes to two large backing files on a dedicated local disk on the emulation machine. The files holding the contents of the emulated disks were the only files on the local disk, only one file/emulated disk was active at once in any given experiment, and all accesses to the backing files passed through the NTFS file system layer but bypassed the buffer cache. The emulation layer added a constant overhead of approximately 510 microseconds to each disk I/O. Compared to a Linux file system on one of the real disks used in our RAID5s, this emulation overhead translates to 10% fewer seeks per second, 41% less

write bandwidth, and 16% less read bandwidth, as measured by the 100MB Bonnie benchmark.

More importantly, we modified the disk emulator to allow the injection of faults into the emulated disk. To make our benchmarks as realistic as possible, it was essential that our set of injected disk faults closely match the types of disk faults seen in practice. To that end, we turned to a study performed as part of the aforementioned Tertiary Disk project at UC Berkeley. Using the 368 disks in the TD array, Talagala recorded the types of faults that occurred over an 18-month period [23]. She found that the most common errors and failures affecting disks included recovered (media) errors, write failures, hardware errors (such as device diagnostic failures), SCSI timeouts, and SCSI bus-level parity errors.

Using this set of errors as a guide, we selected several categories of faults to include in our emulator:

- *correctable media errors* on reads and writes, to simulate disk sectors starting to go bad;
- *uncorrectable media errors* on reads and writes, to simulate unrecoverably-damaged disk sectors;
- *hardware errors* on any SCSI command, to simulate firmware or mechanical errors;
- *parity errors* at the SCSI command level, to simulate SCSI bus problems;
- *power failures* that simulated a disk being disconnected, both during and between SCSI commands;
- *disk hangs* that simulated disk firmware bugs/failures both during and between SCSI commands (these appear as SCSI timeouts to the controller).

All of the faults (except for the fatal ones, like simulating disk power down or infinite timeout) could be inserted either in transient mode, in which case they appeared once then disappeared, or in sticky mode, in which case they continued to manifest themselves once injected. We were particularly interested in the behavior of the software RAID systems in response to the transient faults, as results from Talagala’s TD study indicate that disks rarely fail fast, but rather tend to die slowly with an ever-increasing number of transient and correctable faults [23]. Most availability guarantees made by RAID systems speak only of discrete failures, not of such “fail-slow” failures.

As desired, our set of injectable faults closely matches the set of error conditions seen in the TD array. Note that we were unable to inject one of these types of error condition with our fault-injection harness: the SCSI parity errors at the level of the SCSI electrical protocol. Simulating this type of fault requires either direct access to the wires of the SCSI bus or to low-level registers within the controller, neither of which were available to us.



### 3.2 Configuration of systems under test

We examined three software RAID implementations in our experiments, those shipped with Linux, Solaris 7 Server with Solstice DiskSuite, and Windows 2000 Server. In all cases, the OS and RAID system were installed on a PC with an AMD-K6-2-333 CPU, 64MB of 66MHz ECC DRAM, and a Seagate 5400RPM IDE system disk. Three physical SCSI disks were attached to the machine. Each disk was an IBM DMVS18D 18GB 10000RPM Ultra2-LVD SCSI low-profile drive. Each drive was connected to its own dedicated Fast/Wide (20 MB/s) SCSI bus. Each of the three busses was terminated by either an Adaptec 2940UW controller (set to Fast mode) or one channel of an Adaptec 3940W controller. Note that each drive had a private SCSI bus that was not shared with any other device. A 1GB partition was created at the beginning of each drive for use in the experiments; the remainder of the space on each drive was unused.

The emulated disk (*i.e.*, the PC running the emulation software) was connected to a fourth dedicated SCSI bus on the machine under test using an Adaptec 2940UW controller. Two 1GB emulated disks were created; one was used in the RAID and the other was left as a spare (thus the two were never simultaneously part of the active RAID volume). The backing files for the emulated disks were placed on a dedicated NTFS file system at the beginning of a dedicated IBM DMVS18D 18GB 10000RPM Ultra2-LVD SCSI low-profile drive.

In all cases, unless otherwise noted, the software RAID volume was a 3GB-capacity RAID-5 volume encompassing the three physical disks and one emulated disk. When supported by the implementation, the second emulated disk was used as a spare.

For Linux, we used the Redhat 6.0 distribution with version 0.90-3 of the RAID tools. The RAID volume was configured as 4 active disks plus one spare, left-symmetric parity, and a chunk size of 32. An ext2 file system was used with a 4KB block size and a stripe width of 8.

The Solaris system ran the 3/99 release of Solaris 7 for Intel architectures. We installed version 4.2 of Sun's Solstice DiskSuite and used it to create a RAID-5 "metadevice" with 4 active disks and one spare. The RAID volume was formatted with a Solaris UFS file system with default parameters.

The Windows 2000 Server system was running release candidate build 2128 of the operating system. We used the supplied volume manager to create the RAID-5 logical volume out of 4 active disks. Windows 2000 does not support automatic spares, as far as we could determine, so the spare disks were left as separate dynamic disks in the volume manager. An NTFS file system was used on the array with default parameters.

The three systems were configured as web servers with the documents served from the RAID volume and the logs written to the RAID volume. The Linux and Solaris systems used Apache 1.3.9 as the server, while Windows 2000 Server used the included version of Microsoft's IIS as the server.<sup>1</sup> Other than relocating the logs and document directories to the RAID volume, the servers were left in their default configurations. IIS's performance configuration knob was set to "more than 100,000" hits per day.

### 3.3 Workload generator and data collector

In order to complete our experimental testbed, we needed a source of workload for the web servers running on each OS, and a means of continuously measuring the quality of service delivered by the web servers over time. We chose to use SPECWeb99 [22], a standard web performance benchmark, for both of these tasks. SPECWeb99 uses one or more clients to generate a realistic, statistically reproducible web workload; its workload models what might be seen on a busy major server, and includes static and dynamic content, form submissions, and server-side banner-ad rotation. In each iteration, the benchmark applies a load designed to elicit a certain aggregate bandwidth from the server, then measures the percentage of that bandwidth that was actually achieved. It also measures the number of hits per second delivered by the server and the average response time; we chose to use the number of hits per second (a throughput-oriented performance metric) as the quality of service metric as it was the most tractable and because the other metrics tracked it relatively closely.

We modified the workload generator slightly so that it would fit our model of continuous performance measurement over time: we removed all warm-up and cool-down periods other than the initial warm-up period, adjusted the per-iteration time to 2 minutes, and set the number of iterations to a very large number (manually stopping the generator when the benchmark was complete). This allowed us to obtain performance measurements every two minutes, with each number reflecting the average performance over the previous two-minute period.

We also adjusted the workload generator to reduce the amount of dynamic content from 30% to 1% to keep the disks busy and to avoid saturating the CPU. This restriction was necessary because we used the default high-overhead perl-cgi implementation for dynamic

1. We chose to use IIS for Windows 2000 rather than Apache as we wanted to select the web server that would be typically used with each OS. Since IIS ships with Windows 2000 Server, we believed that it would be the appropriate choice.



content and the CPU on our server testbed was not able to keep up with the higher level of dynamic content.

We configured the applied workload to be just short of the saturation point on each of the three systems by increasing the number of active connections per second (the SPECWeb99 load unit) until a knee was observed in the performance curve, then backing off the load by 5 connections per second. The three systems each saturated at different points, and thus we applied a different level of load to each system in our tests; this accounts for the differences in absolute performance that show up in Figures 2 and 3, below. We chose this load profile instead of applying a consistent load to all three machines in order to isolate the worst-case availability impact on each system. This profile also ensures that we were making fair comparisons between the systems, as some availability behavior (such as RAID reconstruction speed) can be affected by the amount of free system resources. Where pertinent, we also discuss results from experiments in which the applied load was reduced to below the saturation point on each system.

Finally, note that we observed heavy disk activity during the benchmark runs on all three systems, indicating that server-side caching effects were not significant.

## 4 Results

In this section, we present the results of applying our availability benchmarking methodology to the software RAID implementations provided by Linux, Solaris, and Windows 2000. We first look at the single-fault availability microbenchmarks, then move on to study more complex multi-fault availability macrobenchmarks.

### 4.1 Single-fault microbenchmarks

Recall that single-fault microbenchmarks involve injecting a single fault into a running system and observing the resulting behavior of that system without any human intervention. To perform these microbenchmarks for the software RAID systems, we first configured the RAID volume to its nominal state: all disks working, and all spares available. We then started the SPECWeb99 workload generator and allowed it to reach steady state. We next injected a single fault, and allowed the system to continue running (collecting performance data) until the system recovered (performance returned to its steady-state level), stabilized at a different performance level than its steady-state level, or failed. We define failure as not providing service for at least 10 iterations (20 minutes) with no apparent signs of ever returning to service.

In all cases, the faults that we injected were chosen to affect active disk blocks, guaranteeing that the system would be aware of them. By doing so, we avoid injecting so-called *latent* faults, faults that cannot cause failures since they affect only unused data or control paths.

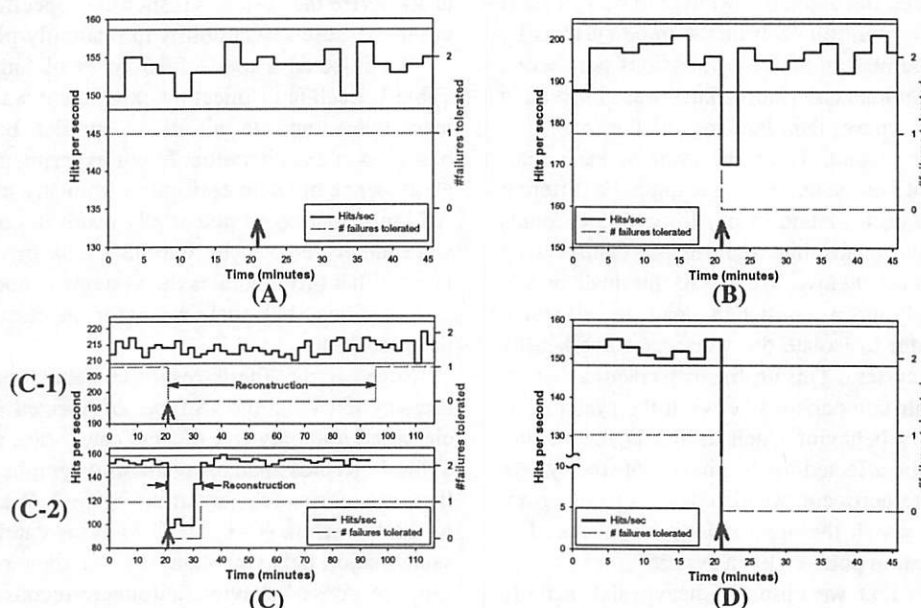
We feel this is a reasonable policy for an availability microbenchmark, as the goal of such benchmarks is to characterize the system's response to specific faults, and not to measure susceptibility to randomly-placed faults.

We injected a total of 15 types of faults, listed in Table 1. Each fault-injection experiment was repeated at least twice, and in all cases, similar behavior was observed in each iteration. In our experiments, we found no evidence of silent corruption from any injected fault. All faults that could potentially result in corrupted data were either detected by the OS's disk driver or RAID layer. What differentiates the systems is not their detection abilities, but their behavior in response to the detected faults.

Surprisingly, these response behaviors across the three systems and the 15 types of injected faults can be classified into only five distinct categories, also listed in Table 1. Representative availability graphs for each of these categories are plotted in Figure 2. We classify two of the behaviors (C-1 and C-2) as subcategories of the same major behavior category, as they represent the same response behavior (automatic reconstruction) but differ in their performance characteristics. Note that each graph in Figure 2 plots the change in two metrics with respect to time. The first metric, represented by a solid line, is the same performance metric discussed above: the number of hits per second delivered by the web server running on the system under test, averaged over two-minute intervals. The second metric, represented by a broken line, represents the minimum number of disk failures the system is theoretically able to tolerate; it is effectively a measure of the system's data redundancy. Note that the graphs also show 99% confidence intervals that were computed from the traces of the systems' normal no-fault performance.<sup>2</sup>

Of the four major categories of observed behavior, the first, A in Figure 2, represents the behavior pattern that occurs when an injected fault has no effect on the RAID system. This graph plots the behavior of the Solaris system in response to a transient, correctable read fault. Notice that the performance curve remains within the confidence intervals despite the injection of the fault; the redundancy measure remains unchanged as well. Effectively, the Solaris system ignores this fault, as it is essentially benign; the disk correctly satisfied the read request, but needed to use ECC bits or multiple reads to obtain the data. Both the Solaris and Windows 2000 systems displayed behavior of this type. Solaris responded this way to all non-fatal faults that we injected, including transient uncorrectable faults (such

2. Analysis showed that the no-fault performance data was normally distributed; thus, the 99% confidence intervals were computed as 2.576 sample standard deviations on either side of the sample mean.



**Figure 2:** Representative availability graphs displaying the five different patterns of behavior observed after injecting faults into the three software RAID systems. Each graph plots two metrics: on the left vertical axis, and represented by a solid line, is the number of hits per second sustained by the web server on the system under test, reported as a single average value over each two-minute interval. On the right vertical axis, and represented by a broken line, is the theoretical minimum number of disk failures the system should be able to tolerate without losing data. Fault injection points are represented by heavy arrows, and 99% confidence intervals for the normal (non-faulty) behavior of the systems are defined by the thin horizontal lines. Table 1 maps each type of injected fault into one of these five behaviors (A, B, C-1, C-2, D) for Linux, Solaris, and Windows 2000.

as a transient, non-repeatable write failure). Windows 2000 behaved similarly, although it was slightly less tolerant of write errors (it did not exhibit this behavior pattern for transient uncorrectable write faults). In no cases did Linux exhibit pattern A—it never transparently tolerated a non-fatal fault.

The second category, B in Figure 2, is more complicated. In this case, the fault is severe enough that the RAID system stops using the affected disk, but is not so severe that the RAID system cannot tolerate it. The performance is slightly affected only during the interval in which the fault was injected, as the system detects and recovers from the fault. The redundancy curve indicates that the faulty disk is no longer used: in this case, the system does not automatically rebuild onto a spare disk, and thus the system cannot tolerate any more disk failures. The particular data plotted in Figure 2(B) is the behavior of Windows 2000 in response to a simulated power failure on one disk of the array (equivalent to physically pulling an active drive from a hot-swap array). This pattern also characterizes Windows's response to other severe faults, including sticky uncorrectable read faults and all uncorrectable write faults.

The magnitude of the performance drop during the fault-injection iteration depended on the type of fault; for uncorrectable writes, it was about 4% of the mean performance, and for power failures, it was about 13% of the mean. Note that the performance drop during the

Type of Fault	Behavior		
	Linux	Solaris	Win2k
Correctable read, transient	C-1	A	A
Correctable read, sticky	C-1	A	A
Uncorrectable read, transient	C-1	A	A
Uncorrectable read, sticky	C-1	C-2	B
Correctable write, transient	C-1	A	A
Correctable write, sticky	C-1	A	A
Uncorrectable write, transient	C-1	A	B
Uncorrectable write, sticky	C-1	C-2	B
Hardware error, transient	C-1	A	A
Illegal command, transient	C-1	C-2	A
Disk hang on read	D	D	D
Disk hang on write	D	D	D
Disk hang, not on a command	D	D	D
Power failure during command	C-1	C-2	B
Physical removal of active disk	C-1	C-2	B

**Table 1:** Classification of system behavior for each of the injected faults. The letters in the rightmost three columns correspond to the pattern of behavior observed after the specified fault is injected, as shown in Figure 2.

fault-injection iteration occurs because the server is near saturation. If we reduce the applied load by just over 20%, the observed performance drops become statistically insignificant. This indicates that Windows is able to trade spare resources for reduced availability impact in certain failure scenarios.

Neither Solaris nor Linux exhibited pattern **B**, as they both support automatic recovery onto a spare disk: when the Solaris or Linux software RAID driver detects a fault severe enough to stop using a disk, it immediately begins reconstructing the data from the failed disk onto the available hot spare. This pattern is illustrated in the graphs labeled **C-1** and **C-2** in Figure 2. **C-1** plots Linux's response to a transient correctable read fault, and **C-2** plots Solaris's response to a sticky uncorrectable write error.

In the Solaris case, we see that the performance curve drops significantly below the lower bound of the confidence interval during the reconstruction period. In contrast, Linux's performance during its entire reconstruction period is statistically indistinguishable from its unperturbed performance. However, Solaris completes reconstruction significantly faster than Linux. The significance of these behavioral differences will be discussed further when we compare the reconstruction behavior of Solaris and Linux with Windows's non-automatic reconstruction in Section 4.2.

Note that during reconstruction, the redundancy curve is not well-defined; the system cannot tolerate a fault to any of the data disks, but it can tolerate a fault to the spare (the destination of the reconstruction).

While Solaris exhibited its version of pattern **C** only for three of the 15 faults (two of which were unquestionably fatal faults), Linux exhibited pattern **C-1** for every injected fault but those falling into pattern **D** even if the fault was transient and non-fatal (like a correctable read).

Finally, the last category, **D**, represents what happens when the RAID system is unable to tolerate the injected fault. As can be seen, the performance drops to zero when the fault is injected; this is usually a result of the RAID driver or operating system hanging. The redundancy curve is not well-defined in this case, since the system is not operational. We observed this type of fault in Solaris, Linux, and Windows when we injected particularly pathological disk hangs in the middle of SCSI command execution.

Table 1 summarizes how the 15 types of injected faults map to the five categories of behavior for each of the operating systems.

**Analysis.** Although limited to a single fault each, these microbenchmark results reveal very interesting facts about the availability guarantees of Linux, Solaris, and

Windows 2000; none of these facts were stated in the documentation supplied with the three systems. Most illuminating are the conclusions that can be drawn about how the three systems treat transient faults. If we exclude the pathological disk hangs and power-failure faults, 8 of the remaining 10 injected fault types simulate transient or recoverable errors that in isolation do not indicate immediate disk failure. Four of these 8 do not even require that the corresponding I/O's be retried. The remaining two faults (sticky, uncorrectable reads and writes) are the only faults in the set of 10 that indicate that the disk is in an unrecoverable state.

Yet for every fault in this set of 10 non-pathological faults, the Linux system exhibited behavior of type **C**, in which the faulty disk is immediately removed from service. In contrast, both Solaris and Windows kept the faulty disk in service on 7 of the 10 non-pathological faults (*i.e.*, 7 of the 8 recoverable errors). Solaris disabled the faulty disk (pattern **C-2**) upon the two unrecoverable faults (sticky uncorrectable reads/writes) as well as on a transient illegal command fault. This behavior is arguably slightly more robust than that of Windows, which disabled the faulty disk (pattern **B**) upon the two unrecoverable errors and a transient uncorrectable write, since an illegal command error typically implies a coding error in the driver or a serious disk firmware error, rather than a potentially transient magnetic glitch.

From these observations, we can conclude that Linux's software RAID implementation takes a totally opposite approach to the management of transient faults than do the RAID implementations in Solaris and Windows. The Linux implementation is paranoid—it would rather shut down a disk in a controlled manner at the first error, rather than wait to see if the error is transient. In contrast, Solaris and Windows are more forgiving—they ignore most transient faults with the expectation that they will not recur. Thus these systems are substantially more robust to transients than the Linux system. Note that both Windows and Solaris do log the transient errors to varying extents, ensuring that the errors are reported even if not acted upon. Windows is more explicit with its reporting, for example visually flagging a disk as "at risk" in the RAID management GUI upon a correctable write error, whereas Solaris relies on the system log for its error recording.

We cannot draw conclusions about a RAID system's overall robustness based solely on its transient-error-handling policy, however. There is another factor that interacts with a system's error handling, and that is its policy for reconstruction. The microbenchmarks demonstrate that both Linux and Solaris initiate automatic reconstruction of the RAID volume onto a hot spare when an active disk is taken out of service due to a fail-



ure. Although Windows supports RAID reconstruction, the reconstruction must be initiated manually, as discussed further in Section 4.2, below. Thus without human intervention, a Windows system will not rebuild redundancy after a first failure, and will remain susceptible to a second failure indefinitely.

The policy choice of automatically or manually-initiated reconstruction interacts strongly with the transient error-handling policy in affecting system robustness. A paranoid RAID implementation without hot spares is very fragile, as it takes only two transient errors to corrupt the RAID volume; likewise, an indifferent RAID implementation has less of a need for hot spares as it will only stop using a disk upon a serious fault. Thus in our case, the non-robustness of the Linux implementation's paranoid approach to transients is mitigated somewhat by its automatic reconstruction, and similarly Windows's lack of automatic reconstruction is partially mitigated by its robustness to transients. Solaris seems to combine the best of both: robustness to transients plus automatic reconstruction upon a fatal error.

Returning to the three systems' transient error policies, if we consider these policies in the context of real failure data, such as that gathered by the Tertiary Disk project, it is clear that none of the observed policies is particularly good, regardless of reconstruction behavior. Talagala reports that transient SCSI errors are frequent in a large system such as the 368-disk Tertiary Disk farm, yet rarely do they indicate that a disk has truly failed [23]. Tertiary Disk logs covering 368 disks for 11 months indicate that 13 disks reported transient hardware errors, yet only two actually required replacement. Those two did not "fail-fast" with head crashes, either: both were replaced due to an excessively large number of transient errors. Additionally, due to the effect of shared SCSI busses and at-times flaky SCSI cabling, at some point over that period every disk in the system was involved in some sort of SCSI error (such as a parity error or timeout) [24]. Even if we ignore these SCSI errors and focus only on the transient hardware errors, Linux's policy would have incorrectly wasted 11 real disks (3% of the array) and potentially 11 spares (another 3% of the array) due to its over-zealous reaction to transient errors. Even worse, if the array did not have enough spares to keep up with the disk turnover, data could have been lost despite the fact that no disk truly failed. Equally poor would have been the response of Solaris or Windows 2000, as these systems most likely would have ignored the stream of intermittent transient errors from the two truly defective disks, requiring administrator intervention to take them offline.

A better RAID implementation would have a more balanced policy for dealing with transient errors. For example, it might be less paranoid initially, tolerating

transient faults until they reached a certain frequency or absolute count, at which point the system would declare a disk dead and stop using it (note that our macrobenchmark experiments showed that neither Windows nor Solaris did this). This kind of policy balances the need for long-term availability (which favors a more relaxed policy) with the fact that disks tend to fail with a stream of transient errors rather than failing fast.

Although none of the RAID implementations we examined is ideal, we can conclude from the microbenchmarks that either Solaris's or Windows 2000's RAID is more suitable for applications requiring high long-term data availability, as both are less likely to fall prey to multiple transient errors (especially in systems that are not closely monitored or conscientiously administered). However, for applications where spare disks are plentiful and short-term availability is most important (*i.e.*, when the performance impact of many transient errors cannot be tolerated, when the system is closely monitored, and when repairs are made quickly), the Linux implementation may be a better choice.

Our results and analysis also argue strongly for the importance of exposing the policy decisions that affect availability in systems like these software RAID implementations. Ideally, the policies would be made configurable, for example by allowing the administrator to select a point on the spectrum between Linux's paranoid response to transients and Solaris's tolerance of them. Doing so would make the policies explicit, and may even simplify maintenance of the system by increasing its predictability, thereby eliminating the need for the administrator to guess at how the system will behave under various conditions.

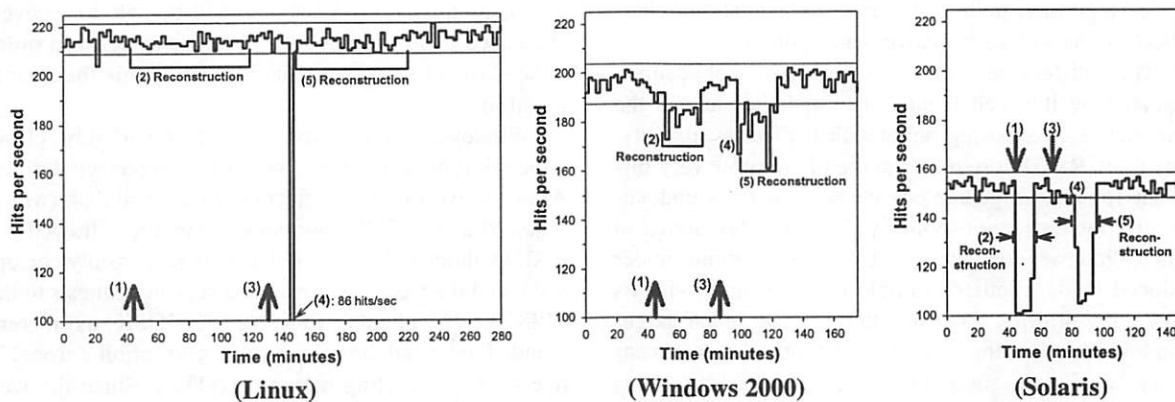
At the very least, availability policies such as those governing the system's response to transient errors should be documented so that administrators and buyers can evaluate the potential robustness of their systems in their particular environment. Until such documentation is commonplace, availability benchmarks such as those described here may well remain the only way to identify and evaluate these important but well-concealed policies.

## 4.2 Multiple-fault macrobenchmarks

After measuring the effects of single failures on the availability of the Linux, Solaris, and Windows software RAID implementations, we next constructed two "fault workloads" designed to mimic real-world scenarios and applied them to the three systems.

**Scenario 1: Reconstruction.** The first scenario includes five events, and models a situation in which a nominally-configured RAID-5 volume with one spare (1) experiences a failure on one of its active disks, (2) is





**Figure 3:** Availability graphs for an availability macrobenchmark with a multiple-fault workload. On the vertical axis, and represented by a solid line, is the number of hits per second sustained by the web server on the system under test. The change in this metric is plotted versus time on the horizontal axis. The thin horizontal lines represent the 99% confidence interval defining the system's normal (no-fault) behavior. The two injected faults are indicated by heavy arrows. The numbers in parentheses on each graph indicate the corresponding part of the fault scenario, as described in Section 4.2. The absolute performance differences between the three systems are due to different applied loads, as described in Section 3.3.

reconstructed (automatically or manually) using the spare, and (3) later experiences a failure on the then-active spare. The scenario is finished by (4) the administrator replacing the two failed disks and (5) reconstructing the volume's redundant data onto one of the new disks. The behaviors of Linux, Windows 2000, and Solaris on this macrobenchmark are plotted in Figure 3. Note that for Windows, we inserted a 6-minute delay to simulate sysadmin response time between detecting the first failure and manually starting the reconstruction. The process of "replacing" the broken (simulated) disks was performed manually, and took approximately 90 seconds in each case.

One obvious difference between the behaviors of the three systems on this benchmark is that Linux and Solaris automatically reconstruct whereas Windows requires human intervention. Most interesting is the difference in reconstruction time between the three systems, and in the performance impact of reconstruction in each case. Linux is the slowest to reconstruct, taking well over an hour each time. However, there is no significant effect on application performance during reconstruction; other than during the time that the disks were being replaced, the performance curve does not fall outside of the confidence interval for normal behavior while reconstruction is taking place.

Solaris defines the opposite extreme. Its reconstruction is over 7 times faster than Linux's, lasting just over 10 minutes. But this speedy reconstruction comes at a performance cost: the web server performance on Solaris is below the lower bound of its normal behavior for the entire reconstruction interval, with a maximum deviation of 34% from its mean no-fault performance.

Windows's behavior is similar to Solaris although not as extreme. Its reconstruction lasts approximately 23

minutes, over twice as slow as Solaris but still more than three times faster than Linux. Windows too shows a performance drop during reconstruction, but it is less significant than Solaris's: the worst-case performance observed was only about 18% below the no-fault mean.

From these observations we can conclude that Solaris and Windows are dedicating more disk bandwidth to reconstruction than is Linux. This again reveals a design tradeoff in the three systems that would be difficult to detect without benchmarks such as these: Linux chooses to emphasize preserving application performance over speedy reconstruction, even though it sacrifices short-term availability. In contrast, Solaris puts a high priority on restoring redundancy despite the performance impact. Windows makes the same tradeoff toward prioritizing reconstruction, but does so less aggressively than Solaris.

Another interesting characteristic of reconstruction is how this behavior changes as the load on the system is reduced. While space constraints prevent us from providing a full analysis in this paper, we did find that at lower loads (such that the systems were unsaturated), Linux and Solaris each exhibited the same reconstruction behavior as in the saturated case in terms of reconstruction time and performance impact. In contrast, Windows was able to decrease both its reconstruction time and the impact of reconstruction on application performance. Our hypothesis is that these behaviors are a function of the scheduling discipline in each of the OSs as well as the priority each system assigns to the reconstruction task. The implication of these behaviors is again significant for availability: Windows seems to be the only system of the three that is able to use the excess resources resulting from lower imposed load to mitigate the availability impact of reconstruction. In

practice, this means that Windows is the only system of the three that can take advantage of hardware with a higher saturation point to improve its availability characteristics as well as its performance potential.

The differences in reconstruction philosophy revealed by this benchmark once again argue for the importance of exposing policies that affect availability. The three RAID systems examined here offer very different robustness guarantees because of their undocumented reconstruction policies. We saw this above in how Windows compared to the other systems under reduced load. Another example is that Linux, with its slow, low-priority reconstruction, has a much larger window of vulnerability to double failures, a weakness exacerbated by its susceptibility to transient errors. This policy is unsuitable if data integrity is most important, and in that case a policy like Solaris's is a better choice. On the other hand, if delivering consistent application performance is more important than preserving the data at all costs, then Linux's policy is reasonable and Solaris's unacceptable. An ideal system would offer the administrator a spectrum of choices between these two extreme policies, but we feel that every system should at least document its chosen policy. Benchmarks such as these offer a convenient tool for doing so.

**Scenario 2: Double failure.** The second scenario mimics a catastrophic failure in RAID systems reported anecdotally by multiple sources. The scenario begins when a nominally-configured RAID volume (1) experiences a disk failure that causes the faulty disk to be removed from service and (2) begins reconstruction (automatically or manually). At that point, (3) the well-meaning system administrator attempts to replace the failed disk, but accidentally pulls out the wrong disk—one of the remaining live disks rather than the dead one; (4) he or she then tries to restore the system to a working state. Removing the live disk should result in a catastrophic failure of the RAID volume, although it did not do so in all cases, as we discuss below. The graphs up to this point are relatively uninteresting, confirming the expected behavior, and are not reproduced here.

What is interesting is the behavior of the systems after the catastrophic failure, and the difficulty of restoring service on the system. We describe this behavior only qualitatively, since in order to quantify it, we would have to find some way of measuring the maintainability of the system, perhaps by modeling the length and complexity of the repair task, which is beyond the scope of this paper.

In this scenario, the last, "catastrophic" failure is actually reversible. According to the RAID availability semantics, the RAID volume should stop serving requests upon a double failure. If the RAID implementa-

tion queues writes to the removed disk while it is unavailable, the administrator could put the disk back in, and theoretically, the system should be able to recover. We tested this hypothesis on the three systems in order to see how close each of them came to this theoretical possibility.

Windows 2000 actually came remarkably close, although it does not queue writes to disconnected disks. After reactivating the accidentally-removed disk (which required a few GUI operations), Windows allowed the RAID volume to be accessed despite its possibly corrupt state, and the web server resumed serving requests to the SPECWeb99 clients. Running CHKDSK as recommended revealed no file system corruption (probably due to the journaling nature of NTFS). Since the web workload was essentially read-only except for the log writes, the only data lost was logging information.

In contrast, we found it impossible to resurrect the Linux RAID volume. The tool used to reintegrate a disk into the volume seemed to only be capable of adding new disks to the volume as spares, which are then automatically used as the target of a reconstruction. There was no obvious way to use the existing tools to convince Linux that the replaced disk contained real data. Therefore, the only way to resurrect the volume was to recreate and reformat it, then restore data from backup.

Solaris demonstrated radically different behavior than the other two systems. Unlike the other systems, it did not disable the RAID volume after the double failure: it kept the array active with the two still-functioning disks and the partially-reconstructed spare. This behavior violates the availability semantics of RAID-5, since at this point a large portion of the data is missing (any data that had not yet been reconstructed on the spare is permanently lost). By keeping the RAID array active and using the nonsensical data on the partially-reconstructed spare, Solaris allows applications to read garbage data. In our case, this was manifested by the web server returning garbage to the SPECWeb client and via numerous UFS file system corruptions as reported by fsck. Furthermore, when we plugged the accidentally-removed disk back in, Solaris was happy to automatically switch back to using it to service I/Os, deactivating the partially-reconstructed spare. However, because Solaris had continued to use the array while the second disk was removed, the data on that disk was significantly out-of-date and the file system was corrupted as a result of reinserting it.

We believe that Solaris's behavior is absolutely incorrect for a RAID system. A RAID system should not fabricate data to maintain availability unless explicitly requested to do so, *i.e.*, by manually forcing the reactivation of a reinserted disk, as with Windows. Furthermore, we were not able to find any mention of this

behavior in any of the Solaris documentation, which again argues for the importance of benchmarks like these to expose the undocumented availability policies in systems like these software RAIDs.

Thus in this scenario, Solaris clearly loses due to its willingness to transparently serve up garbage data. But Windows 2000 wins on maintainability, as its robust file system and flexible RAID implementation allows the opportunity for at least some use of the RAID volume to continue servicing user requests while the system is being restored from backup (but only at the explicit request of the administrator, unlike Solaris). Although this may not always be the best thing to do, Windows provides the ability should it be desired.

In this second macrobenchmark, we have the beginnings of a framework for a combined availability and *maintainability* benchmark—the fault injection workload for this scenario brings the system to a state in which maintenance is required; to complete the benchmark, we would use a quantitative maintenance model to simulate repair of the system, then use that data to complete the availability graph for this scenario. We are currently pursuing this as future work.

## 5 Related Work

The notion of benchmarks to measure system availability or “robustness,” although perhaps not familiar to the systems community, has not been neglected by the fault-tolerance community. Siewiorek describes “robustness benchmarks” based on fault injection performed primarily by using an application to feed corrupt input to the system [21]. Tsai, working on Tandem machines, proposes another set of reliability benchmarks based on software-implemented fault injection and a synthetic workload generator. His metrics include an average measure of performance degradation due to faults, a primitive version of our time-dependent quality of service metrics [25]. Koopman describes benchmarks to test OS robustness by feeding corrupt data to system calls [17]. The major difference between these benchmarks and the ones we propose is in their goals and the knowledge they assume. Tsai’s and Siewiorek’s benchmarks are primarily designed to test particular known fault-tolerance mechanisms deployed in fault-tolerant hardware and software systems; to this end, their benchmarks target and evaluate specific components, layers, or mechanisms in the system under test, and thus assume knowledge about the error-detection mechanisms and general structure of that system. In contrast, our benchmarks take a more black-box approach, assuming little about the system under test (not even that it is fault tolerant), and applying faults designed to match real-world failure patterns. Koopman’s benchmarks do this as well, but are limited to faults generated

by passing corrupt data to system calls; we try to mimic more general faults, including hardware failures.

An additional key difference is that our benchmarks measure the system’s availability behavior in terms of application-specific metrics that reflect quality of service visible from the client’s point of view. Finally, our multi-fault workloads go beyond the isolated faults examined by Siewiorek, Tsai, and Koopman by relating the behavior of a system to realistic scenarios that affect large-scale server systems and by providing a foundation for the expansion of the benchmarks to incorporate the measurement of maintainability.

The techniques of fault injection that we use are also not uncommon in the fault-tolerance community, where fault injection is commonly used in a case-specific manner to verify fault tolerant systems, to generate models of fault tolerance behavior, and to study fault propagation [1][5][6][8][16][18]. However, most of this work uses either very low-level hardware fault injection that requires expensive and dangerous equipment (such as heavy-ion bombardors) [5], or software-implemented fault injection. The former is not tractable for general use because of the cost and complexity, and the latter is not particularly portable, as it generally requires modifications to the OS or driver layer. In contrast, our approach of hardware fault injection at standard interfaces (such as the SCSI-level fault injection used for the RAID study) is both portable and relatively simple; for example, we could have easily used the same fault-injection setup (consisting of off-the-shelf PC hardware and software) to measure the availability of software RAID on a SPARC/Solaris machine.

Finally, there have been several studies of RAID reliability and availability [14][15], but these have focused on simulation studies of hardware RAID, and none have examined RAID in the context of a general availability benchmark.

## 6 Future Directions

We are currently pursuing several extensions of the work in this paper. First, we are planning to expand our experience with the availability benchmarks by applying them to more complex systems, such as database management systems. We are also working on a general framework for maintainability benchmarks, and in particular are looking into ways to model the behavior of a human administrator. We are also expanding the fault-injection capabilities of our testbed to include the capability of inserting memory faults and OS driver faults. Finally, under the umbrella of the ISTORE project, we are building the ISTORE-1 prototype, an 80-node cluster system that incorporates custom fault-injection and diagnostic hardware that should enable the extension of this work to distributed systems and applications.



## 7 Conclusion

In this paper we have laid out the framework for new kinds of benchmarks in an area left relatively unexplored by computer science researchers: availability. We demonstrated the efficacy of our general availability benchmarking methodology by specializing it to the study of software RAID systems, and by then using it to unearth insights into the behavior of the Linux, Solaris, and Windows 2000 software RAID implementations. In particular, we were able to uncover each system's (undocumented) policy for mapping transient faults into failure conditions, and to quantify the impact of these policies and of the systems's failure recovery policies on the quality of service and availability delivered by I/O-intensive applications running on those systems.

While we believe that the power of our approach is clearly illustrated in these insights, this paper is only a first tentative step down what surely must be a long road to the important goal of comprehensive, portable, and meaningful benchmarks for availability, maintainability, and evolutionary growth. We feel that reaching that goal is crucially important for the field, and we look forward to companionship on this journey.

## Acknowledgments

This work was supported in part by ARPA grant DABT63-96-C-0056. The first author was supported by a Department of Defense National Defense Science and Engineering Graduate Fellowship. The germ for many of the ideas in this paper came out of discussions with members of the ISTORE group at UC Berkeley, and in particular with David Oppenheimer. We also wish to thank IBM for donating the disks used in these experiments, Andataco (and particularly Darryl Keiser) for providing extra drive enclosures on very short notice, Bill Casey of ASC for fixing the last bugs in the disk emulation library, and both the anonymous reviewers and members of the ISTORE group for their feedback. Finally, the first author thanks Randi Thomas for the encouragement to make this paper a reality.

## References

- [1] J. Arlat, A. Costes, et al. Fault Injection and Dependability Evaluation of Fault-Tolerant Systems. *LAAS-CNRS Research Report 91260*, January 1992.
- [2] R. Arpaci-Dusseau. Performance Availability for Networks of Workstations. *Ph.D. Dissertation*, U. C. Berkeley, December, 1999.
- [3] ASC, Inc. Advanced Storage Concepts VirtualSCSI library. <http://www.advstor.com/vscsi.html>.
- [4] A. Brown, D. Oppenheimer, et al. "ISTORE: Intropective Storage for Data-Intensive Network Services." *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, Arizona, March 1999.
- [5] J. Carreira, D. Costa, and J. Silva. Fault injection spot-checks computer system dependability. *IEEE Spectrum* 36(8):50-55, August 1999.
- [6] S. Chandra and P. Chen. How Fail-Stop are Faulty Programs? In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing*, June 1998.
- [7] P. Chen, E. Lee, et al. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys* 26(2):145-185, June 1994.
- [8] R. Chillarege and N. Bowen. Understanding Large System Failure—A Fault Injection Experiment. In *Proceedings of the 1989 Fault-Tolerant Computing Symposium (FTCS)*, 356-363, 1989.
- [9] Forrester. <http://www.forrester.com/research/cs/1995-ao/jan95csp.html>.
- [10] A. Fox, S. Gribble, et al. Cluster-Based Scalable Network Services. In *Proceedings of SOSP '97*, October, 1997, St. Malo, France.
- [11] Gartner. <http://www.gartner.com/hcigdist.htm>.
- [12] J. Gray. Locally served network computers. Microsoft Research white paper. February 1995. Available from <http://research.microsoft.com/~gray>.
- [13] J. Hennessy. The Future of Systems Research. *IEEE Computer* 32(8):27-33, August 1999.
- [14] Y. Huang, Z. Kalbarczyk, and R. Iyer. Dependability analysis of a cache-based RAID system via fast distributed simulation. In *Proc. 17th IEEE Symposium on Reliable Distributed Systems*, October 1998.
- [15] M. Kaâniche, L. Romano, et al. A Hierarchical Approach for Dependability Analysis of a Commercial Cache-Based RAID Storage Architecture. In *Proceedings of 28th International Symposium on Fault Tolerant Computing*, June 1998.
- [16] W. Kao, R. Iyer, and D. Tang. FINE: A Fault-Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults. *IEEE Trans. Software Eng.*, 19(11):1105-1118, November 1993.
- [17] P. Koopman, J. Sung, et al. Comparing Operating Systems Using Robustness Benchmarks. In *Proceedings of the 16th Symposium on Reliable Distributed Systems*, 72-79, October 1997.
- [18] W. Ng and P. Chen. The Systematic Improvement of Fault Tolerance in the Rio File Cache. In *Proceedings of the 1999 Symposium on Fault-Tolerant Computing*.
- [19] M. Richtel. Keeping E-Commerce On Line; As Internet Traffic Surges, So Do Technical Problems. *The New York Times*, 21 June 1999.
- [20] M. Satyanarayanan. *Digest of Proceedings, Seventh IEEE Workshop on Hot Topics in Operating Systems (HotOS-VII)*, March 29-30, 1999, Rio Rico, AZ.
- [21] D. Siewiorek, J. Hudak, et al. Development of a Benchmark to Measure System Robustness. In *Proceedings of the 1993 International Symposium on Fault-Tolerant Computing*, 88-97, June 1993.
- [22] SPEC, Inc. *SPECweb99 Benchmark*. <http://www.spec.org/osg/web99>.
- [23] N. Talagala. Characterizing Large Storage Systems: Error Behavior and Performance Benchmarks. *Ph.D. Dissertation*, U. C. Berkeley, September, 1999.
- [24] R. Thomas, N. Talagala. What Happens Before a Disk Fails. Talk at the Winter 1999 IRAM Semi-annual Research Retreat. January, 1999.
- [25] T. Tsai, R. Iyer, and D. Jewett. An Approach towards Benchmarking of Fault-Tolerant Commercial Systems. In *Proceedings of the 1996 Symposium on Fault-Tolerant Computing (FTCS)*, June 1996.



# Performing Replacement in Modem Pools

Yannis Smaragdakis

*College of Computing  
Georgia Institute of Technology  
Atlanta, Georgia 30332  
yannis@cc.gatech.edu*

Paul Wilson

*Department of Computer Sciences  
The University of Texas  
Austin, Texas 78712  
wilson@cs.utexas.edu*

## Abstract

We examine a policy for managing modem pools that disconnects users only if not enough modems are available for other users to connect. Managing the modem pool then becomes a replacement problem, similar to buffer cache management (e.g., in virtual memory systems). When a new connection request is received, the system needs to find a user to “replace”. In this paper we examine such demand-disconnect schemes using extensive activity data from actual ISPs. We discuss various replacement policies and propose CIRG: a novel replacement algorithm that is well suited for modem pools. In general, the choice of algorithm is significant. A naive algorithm (e.g., one that randomly replaces any user who has been inactive for a while) incurs many tens of percent more “faults” (i.e., disconnections of users who are likely to want to be active again soon) than the LRU algorithm, which, in turn, incurs 10% more faults than CIRG. For good replacement algorithms, the impact can be significant in terms of resource requirements. We show that the same standards of service as a system that does not disconnect idle users can be achieved with up to 13% fewer modems.

## 1 Introduction

A pool of modems is a time-shared resource: there are typically many more potential users than can simultaneously connect. The most common instance of modem pools is in telephone-modem-based Internet Service Providers (ISPs), where modems accept user connections over the telephone network. When all modems are occupied, no more connections are allowed and the users attempting to connect receive a busy signal. Although ISPs strive to avoid busy signals, this is not always feasible. The most common line of defense is to keep a fixed ratio of subscribers to modems, with a value of 10:1 sometimes considered safe for avoiding busy signals. This policy, however, is hard to maintain consistently

(e.g., as usage patterns change in response to marketing actions<sup>1</sup>) and cannot always protect fully against busy signals. As an added measure, some ISPs try to discourage subscribers from constant modem use by setting usage limits and applying surcharges for exceeding them. An additional widespread practice is to proactively disconnect users who have been idle for some fixed time interval or users who have been connected continuously for some period of time. Recently, Douglass and Killian [DK99] improved over fixed idle timeout policies with *adaptive* proactive disconnections of users. Their technique varies the idle time threshold for disconnecting a user, based on the user's past activity patterns.

Although all of the above techniques are valuable for certain scenarios, they do not acknowledge that, in the most common case, the cost of allowing a user to stay connected is a function of the current load. ISPs typically suffer a cost for prolonged usage *only* when the modem pool utilization has reached capacity.<sup>2</sup> In other words, it is commonly of no cost to ISPs to allow users to stay connected when modems are available. The cost of telephone lines is usually fixed for ISPs, regardless of usage. Also, the cost of local phone calls in the US is commonly fixed (or zero) for users, regardless of call duration. To the best of our knowledge, the only policy in use that somewhat relates usage limits and current load is the variable timeout policy: some ISPs have shorter timeouts for pre-defined “peak hours” (e.g., 6pm to midnight). Nevertheless, more sophisticated schemes are easy to implement and, as we argue, inconvenience users much less.

In this paper, we examine the case of treating a modem

<sup>1</sup>In one well-publicized case, a change in America Online usage pricing caused many users to stay connected longer, allegedly resulting in a barrage of busy signals that prompted a lawsuit by dissatisfied customers [Aho97].

<sup>2</sup>This is not to say that ISPs have no incentive for limiting usage even when capacity is not reached. For instance, ISPs could limit usage expecting that the market will support higher prices for increased usage. This is orthogonal to the concerns of this paper.

pool as a replacement buffer. That is, users are not disconnected until all modems are occupied. When a new user attempts to connect with all existing modems occupied, there are two possible outcomes: either there are no “idle” users currently and the new user will be denied service (busy signal) or one of the idle users will be *replaced* (i.e., disconnected in favor of the new user). Strictly speaking, this approach is not easy to implement exactly, because busy signals are generated by the telephone network, not the ISP. Nevertheless, as we will explain, the policy can be easily approximated closely in an actual modem pool.

The interesting question in replacement scenarios is how to choose the user to replace. We examine three different replacement algorithms: LRU, CIRG, and RANDOM. The LRU algorithm replaces the user who has been inactive the longest. The CIRG algorithm (for Conditional Inter-Reference Gap) is novel and is inspired by Phalke's work on inter-reference gaps for access prediction in virtual memory systems [Pha95]. In intuitive terms, CIRG attempts to recognize access patterns for individual users and should be a good fit for the modem pool domain. Finally, the RANDOM algorithm selects any idle user for replacement. In Section 3 we discuss in detail the replacement problem for modem pools. The problem is related but different from replacement in other settings (e.g., virtual memory systems). The differences in the problem and in the expected access patterns (e.g., no *spatial* locality) guide the design of replacement algorithms and we explain what the characteristics of a good replacement algorithm should be.

For our experiments we used three extensive traces of user activity, which cover several distinct circumstances. The results of our simulations show that our approach is much less inconvenient to users than proactive disconnections after a fixed period of inactivity. Similar to the Douglass and Killian work [DK99], our metric for “inconvenience” counts “faults”, that is, disconnections of users who are likely to want to be active again soon. In these terms, disconnecting users only when the modem pool is full results in an order of magnitude fewer faults than a fixed idle timeout policy. More interestingly, however, we show that the choice of replacement algorithm is important. RANDOM has a much higher cost (by over 20% and up to 1000%) than LRU. LRU, in turn, performs worse than CIRG, often by 10% or more. (Both LRU and CIRG turn out to be very good predictors of future idle times.) This shows that the naive approach of disconnecting any idle user when the load is high is far from ideal.

It is an interesting challenge to further quantify the benefits of our approach (or of any other approach to dis-

connecting idle users). The difficulty is that disconnecting users may encourage them to set up their connection so that it appears to be permanently active. For this reason, although we present results for a wide range of values, we concentrate on a range that guarantees a quality of service similar to that of the trace collection environment. (The natural assumption is that when users are not annoyed, they will not change their behavior.) We show that, for CIRG, good quality of service can be maintained with up to 13% fewer modems than a no-disconnections policy. This result shows that when there are not enough modems (e.g., due to a surge in subscriber numbers) the impact can be softened significantly using our technique.

Our experiments are interesting, however, regardless of the actual economics and current practices in the ISP business. What we are really demonstrating is the kind of activity patterns that the combination of human users and modern Internet tools (e.g., browsers, email clients, etc.) is expected to exhibit. In particular, we analyze which techniques work well for predicting future idle times. These results may be applicable to more contexts than telephone-modem-based ISPs—practically in every case a time-shared resource is accessed interactively by Internet users and we want to predict future inactivity times.

## 2 Studied Workloads

We describe early on the workloads we studied, so that we can use concrete examples in the subsequent discussion.

The first of our traces is the one collected by Douglass and Killian and used in their study [DK99]. This trace consists of the activity over a week in May 1998, polled every 30 seconds, of 100 users of a modem pool maintained by AT&T Labs. The modems in the pool served as the uplink of an asymmetric connection (the downlink was a cable connection). The setup where this trace was collected is interesting because it uses static IP addresses. This makes disconnects less undesirable: all TCP connections can stay open until the next modem connection. Thus, the cost of disconnecting a user is primarily in terms of the delay of reconnection. As a consequence, the modem pool itself has an aggressive 15 minute idle timeout (which would probably be too short for general ISPs). A few users have no inactivity timeout (after their request) and can stay connected indefinitely. An interesting characteristic of this setup is that all users have a dedicated phone line. A direct consequence is that users cannot be denied service but also that they have less of an incentive to disconnect explicitly. In fact, the trace we studied contains no disconnection information. Douglass and Killian report that 15 users appeared active at all times. (Actually this number depends on what is

defined as “at all times”: for a higher inactivity tolerance, 19 users appear active at all times.)

Our other two traces are from Telesys—the Internet Service Provider for the University of Texas community. In September 1998, Telesys was the largest ISP in the Austin, Texas metropolitan area (we are not aware of more recent data) with over 3,000 modems, serving over 34,000 subscribers [Aca98]. (The total number of subscribers seems to be artificially high: unusually many subscribers did not use their accounts at all during the two periods of our study. The inflation in the number of subscribers is probably due to the low cost of Telesys for its restricted user base—many Telesys users can afford to subscribe even though they rarely use the service.) Our Telesys traces contain explicit disconnect information, and users are allowed to stay connected indefinitely. (Telesys does not disconnect idle users despite a stated policy of a 2-hour idle timeout.) Telesys has not had busy signals due to limited capacity in the past few years [Aca98], hence no users were denied service during our trace collection.

We collected the two traces of Telesys activity by repeatedly polling the modem servers in two-minute intervals. The traces are from periods of significantly different activity. The first was collected in a period of low activity (June 26 to July 6, 1999, which includes the 4th of July holiday) with 18,086 distinct users accessing the system in this time period. The maximum number of simultaneously connected users was 2,151, and 5 users were connected at all times. In all, there were over 315,000 connections (i.e., instances of users connecting and disconnecting). The second trace was collected in a high activity period (November 1 to 8, 1999). 21,221 distinct users accessed the system in that time with a maximum of 3,024 users connected simultaneously. There were over 347,000 connections in total and 11 users stayed connected throughout the week-long tracing period. The number of connected users as a function of time for the Telesys traces is plotted in Figure 1.<sup>3</sup>

An interesting issue concerns users who stay permanently connected (termed “workaholics” in the Douglass and Killian study, with the term originating in Barabási and Imieliński [BI94]). If “workaholics” are connected regardless of disconnections (e.g., due to periodic tasks using the network) they should be included in the study. If they are active only because they try to “beat” the disconnection policy of the tracing environment, they should be excluded. This issue is significant only for our first trace (from AT&T Labs) because of the idle time-

<sup>3</sup>No similar data can be plotted for the AT&T Labs trace since no disconnection information is available. The set of connected users for that trace depends on the disconnection policy used.

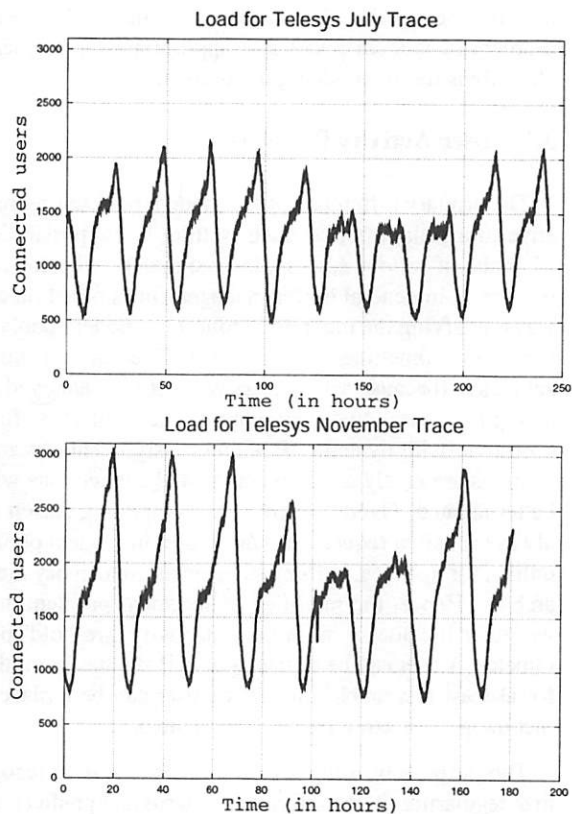


Figure 1: Number of connected users as a function of time. The first plot shows three “low usage” days because of the 4th of July holiday.

out that was in place when the trace was collected, and because of the large percentage of “workaholics”. In the Douglass and Killian study, “workaholics” were excluded from the experiments. The reason was that after interviews with individual users, Douglass and Killian concluded that their constant activity was due to programs used as a response to the 15-minute idle timeout. (That is, these users ran periodic tasks to simulate activity and avoid automatic disconnections.) Similarly, we excluded “workaholics” from our first trace. Nevertheless, we have also performed all experiments with the full trace and concluded that the inclusion of “workaholics” does not fundamentally affect our results or our conclusions (absolute counts may change but relative differences are practically the same).

### 3 Replacement Algorithms for Modem Pools

The replacement problem in buffer management is a general and well-studied problem. In this section we



identify the special characteristics of the replacement problem in modem pools and appropriate replacement algorithms for the modem pool domain.

### 3.1 User Activity Patterns

The primary difference of the modem pool setting relative to a general replacement setting is the possibility of denial of service (that is, busy signals). Whereas replacement in general buffer management is based on always satisfying the incoming request, in modem pools it is better to sometimes deny requests. The reason is both subjective (because active users will get very annoyed at losing a resource that they claimed first) but also often objective: with dynamic IP address assignment, disconnections are costly as all existing TCP connections will be terminated. Overall, there is no compelling reason to always satisfy a request for connection in modem pools, unlike, for instance, buffer management in memory hierarchies. Hence, the modified replacement problem that we study, includes a "minimum inactivity threshold" parameter. A user can be replaced only if she has been idle for at least this much time. If no user can be replaced, incoming connection requests are denied.

The purpose of a replacement algorithm is to recognize regularities in the reference patterns and predict correctly which entities can be replaced with minimum cost. Thus, the unique characteristics of the modem pool domain influence the choice of a replacement algorithm. A characteristic of this domain is that there is no concept of *spatial locality*, or, in general, any way to associate the activity of two different users. "Spatial locality" refers to the observation that once an entity becomes active, other entities that are close to it in a well-defined space are likely to also become active soon. For instance, in virtual memory systems once a page is referenced, pages next to it, either in the address space, or in the recency space,<sup>4</sup> are also likely to be referenced soon. No such inference can be drawn in the case of modem pools. Individual users are quite likely to have no interactions with one another and there is rarely reason for their inactivity patterns to be correlated. The lack of spatial locality means that replacement algorithms that may be successful in other domains are unsuitable for modem pools. For example, recently proposed algorithms for virtual memory replacement, like SEQ [GC97] (which detects regularities in the address space) or EELRU [SKW99] (which detects regularities in the recency space) are not appropriate for modem pool replacement.

The question then becomes, what kind of regularities are there in modem activity? Clearly, activity patterns are

<sup>4</sup>The recency space is the total ordering of pages according to how recently they were last referenced.

mainly dictated by human users but also Internet tools (e.g., email clients, browsers, etc.). It is interesting to examine whether strong regularities exist. For this, we consider the distributions of idle times before a user becomes active for the three traces of our study, shown in Figure 2.

There are a few observations we can make. First, as expected, there is strong *temporal locality* exhibited by all traces: the number of times a connection is re-activated generally decreases very rapidly as a function of the idle time. A second observation is that there is evidence of programmatic (i.e., periodic) behavior in all traces. In the AT&T Labs trace, strong spikes of activity appear at 5 and 10 minutes of idle time. Additionally, strong activity is observed after 14 minutes of idle time, which is immediately before the inactivity timeout of 15 minutes for this service. Further activity, however, is evident for idle times (16 minutes) slightly *longer* than the disconnection timeout. This suggests that some users may have set their systems to reconnect immediately after a non-user-initiated disconnection. This is one of the features that may interact with our simulations and are discussed further in Section 4.1. The Telesys traces also exhibit periodic activity, for instance, every 10, 20, and 30 minutes. Activity after exactly 5 or 15 minutes of idle time could also be strong for the Telesys traces, but due to the trace granularity (2 minutes) it cannot be distinguished very well.

We see, therefore, that the AT&T Labs and Telesys workloads show evidence of both human and programmatic activity. Recall that "workaholics" (i.e., users who tend to be permanently connected) were excluded from the AT&T trace. Hence, it is reasonable to believe that the periodic activity observed is not just a response to the inactivity timeout of the system. Such periodic activity is significant (for instance, periodic stock price updates are often more important to home traders than any interactive traffic).

Overall, and quite expectedly, Figure 2 shows that human-produced activity has excellent temporal locality: a connection receiving only human input tends to stay inactive once it becomes inactive. Most of the activity seen after some inactivity seems to be programmatically produced because it coincides with easily identifiable time intervals. A good replacement algorithm should be able to perform well for both kinds of activity.

### 3.2 The CIRG Algorithm

The CIRG replacement algorithm (for *Conditional Inter-Reference Gap*) is a novel (to our knowledge) algorithm, loosely based on the Inter-Reference Gap model proposed by Phalke [Pha95]. The term "inter-reference



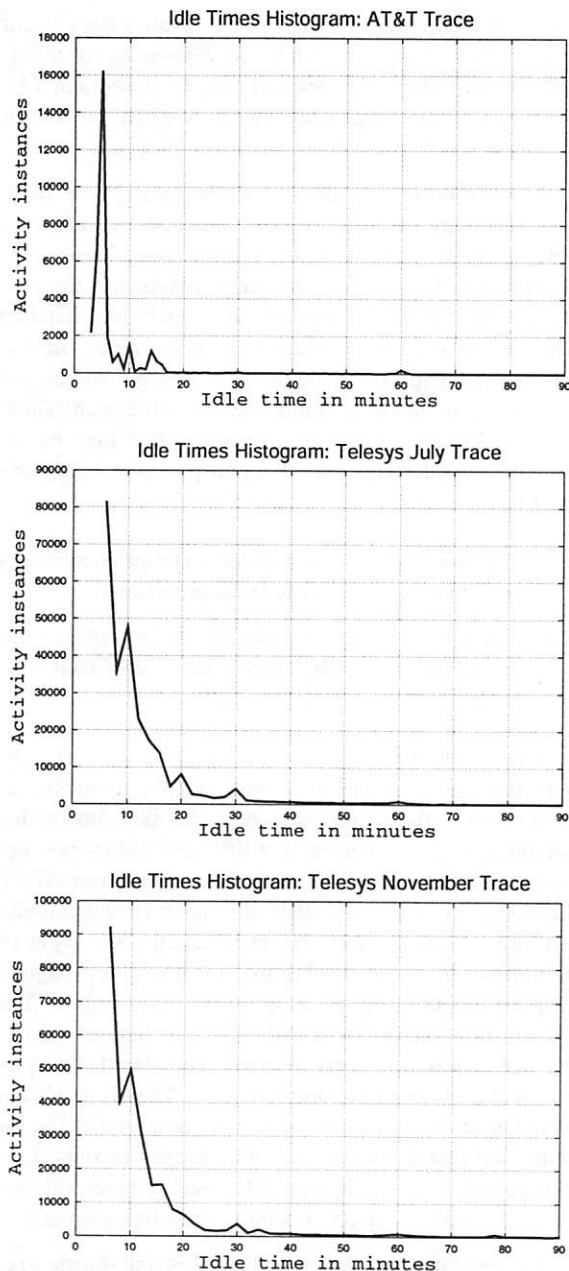


Figure 2: Idle time distributions for the three traces of our study. The left ends of all traces rise quickly and are excluded for clarity.

gap” is a synonym for idle time in the user activity domain. The CIRG algorithm keeps per-user information about the most recently encountered idle times and uses it to predict future idle times. In particular, for each user, CIRG keeps a list, which we call *idle*. (We will later generalize this to multiple lists.) The first element of the list, *idle*(1) is the user's idle time before the user last became active. For  $i > 1$ , the  $i$ -th element of the list, *idle*( $i$ ), is the total idle time for a user the last time the user was idle longer than *idle*( $i - 1$ ) time units. For example, the contents of the *idle* list could be:

2, 5, 13, ...

(with minutes as the time unit) meaning that the user was idle for 2 minutes before she last became active. The last time the user was idle for more than 2 minutes, she stayed idle for 5 minutes. The last time the user was idle for more than 5 minutes, she stayed idle for 13, and so on.

Using the *idle* list and the current idle time, the CIRG algorithm computes how long a user stayed idle the last time she was idle for as long as now.<sup>5</sup> This is used as an estimate of the total idle time until a user will next become active. In our example, given a current idle time of 7 minutes, CIRG will predict that the user will stay idle for another 6 minutes (for a total of 13). The user with the highest predicted future idle time is the one to be replaced.

This scheme can easily be generalized to multiple lists.<sup>6</sup> That is, with  $k$  lists, CIRG can compute how long a user stayed idle the last  $k$  times she was idle for as long as now. A reasonable estimate of future idle time will then be the average of the predicted values from each of the  $k$  lists. To keep the list lengths short, we can quantize the idle times values (e.g., by rounding all idle times to the closest multiple of 2 minutes).

We have experimented with the CIRG algorithm in several different replacement settings. Overall, CIRG is, as expected, good for detecting regular patterns in behavior. CIRG is not an ideal algorithm for some settings (e.g., virtual memory replacement) because it keeps

<sup>5</sup>If the user has never been idle for as long as now, a reasonable guess for the total idle time is twice the current idle time (i.e., an LRU-like estimate can be employed: the user's predicted future idle time is the same as her past idle time).

<sup>6</sup>The procedure for maintaining  $k$  lists, *idle*<sub>1</sub>, ..., *idle* <sub>$k$</sub>  is simple but we outline it here for completeness. Each list has an update operation that takes an idle time parameter. The update operation for the entire data structure (all lists) calls the update operation for the first list, *idle*<sub>1</sub>, with parameter  $t$ , when the user becomes active after being idle for time  $t$ . The update operation for list  $i$  examines all elements of list *idle* <sub>$i$</sub>  in order from the beginning of the list. For each element  $e = \text{idle}_i(j)$ , if  $e < t$ , the element is removed from list  $i$  and if  $i < k$ , the update operation is called on list  $i + 1$  with parameter  $e$ . Finally,  $t$  is inserted in list *idle* <sub>$i$</sub> .

statistics in terms of time differences between accesses. (As we explain in [SKW99], inter-reference time is not a reliable metric of locality for modern programs because they may access vastly different amounts of memory in the same amount of time.) Nevertheless, we expected CIRG to be appropriate for the modem replacement domain, which deals with human users and real-time idleness.

An interesting aspect of CIRG is that, because it keeps statistics per user, it can recognize an arbitrary number of different periodic patterns. Thus, CIRG is a good algorithm for dealing with *multiple* different regular patterns. Such patterns could occur because of Internet tools pulling information (e.g., email clients downloading messages every 10 minutes, browsers updating the displayed stock prices page every 5 minutes, etc.). Patterns could also occur because of user habits (e.g., the user usually takes 5-10 minute breaks).

## 4 Experimental Setup

Given the activity traces for the studied workloads, we simulated different disconnection policies and modem pool sizes. In this section, we discuss the assumptions behind our experiments and the metrics used to evaluate different approaches.

### 4.1 Validity of Simulations

Our experiments consist of simulations of modem pools with different numbers of modems and different disconnection policies than those in place during tracing. Nevertheless, the actual user activity might have been different if the actual simulated policies were in effect. For instance, one common user reaction to strict idle timeout policies is to use programs that simulate activity so that the connection appears to be always in use.

This interaction of policies and user decisions is something that no predictive study involving human users can compensate for. Nevertheless, although this issue can certainly affect the exact results of our experiments, it should not affect the overall picture and our conclusions. There are two reasons for this. First, the most important of our observations are made under conditions that guarantee a quality of service similar to that of the tracing environment: busy signals are extremely rare and disconnections of users who want to be active again soon are very few. The second reason is more subtle but very important. The most likely user reaction (if any) to inconvenient disconnections is to simulate constant activity so that no disconnection occurs. This penalizes all disconnection policies equally per user (the user simply cannot be disconnected, and occupies a modem as long

as she wants). Thus, a policy that causes more user inconvenience can be expected to be penalized more than a policy that causes less inconvenience (because more users will attempt to avoid disconnections in the former case). Therefore, even though the absolute values of our metrics may change, their *relative differences* for different disconnections policies will only be *accentuated* by the user's conscious choices. Hence, we believe that our results will hold true when actually employed.

Apart from the case of a conscious user choice, however, there are cases when programmatically generated activity may interact with our simulations. This means that the actual behavior might not be identical to the simulated one if the studied workloads were indeed handled with the disconnection policy under simulation. The differences are expected to be insignificant, but we discuss them here so that our assumptions are in the open. There are two kinds of systematic activities that may be affected by a different disconnection policy and may appear in our traces.

1. The user may have set her dial-in program to reconnect on non-user-initiated disconnections.
2. The user may have set her dial-in program to *not* dial in automatically every time an application attempts to make a TCP connection.

The potential problem in case 1 is dual. First, the observed behavior in our trace may include automatic reconnections after disconnections. This is behavior that would not have occurred if a different policy had not caused the disconnection. This affects only our AT&T Labs trace, which was collected in an environment with a 15 minute idle timeout. Nevertheless, the activity at 16 minutes of idle time (see Figure 2) is not high enough to suggest that this may be a significant interference. The second possible problem with case 1 is that new activity will appear after every disconnection, but this activity is not evident from the original trace. Therefore, all our simulations should be viewed as accurate under the assumption that users do not automatically reconnect on disconnect. Even in the opposite case, however, this behavior is likely to affect all simulated policies equally.

The potential problem with case 2 is that during tracing we may have missed some behavior because of the disconnection policy that was in place in the tracing environment. That is, the user might have been active, had she not been previously disconnected. (The issue clearly is relevant to programmatic activity, not interactive user activity—the user would explicitly connect if she wanted to use the network interactively.) This should not be a problem—if the behavior was not important enough to occur in the traced system, it was probably not important

enough to occur in any simulated systems. It should also be noted that our traces reflect all activity that originated from a user. We have no way of telling if some of this activity would not have occurred if the user had previously been disconnected. Hence, our simulations represent the behavior of a system where users do dial in automatically every time they are disconnected and an application tries to access the network (e.g., every time their email client needs to download messages).

## 4.2 Metrics

Our first metric of performance of a disconnection policy is, expectedly, the number of busy signals that the policy incurs for a given number of modems. Nevertheless, this number depends primarily on the number of users who can potentially be disconnected and the latter number is almost the same for all replacement policies. Our approach is to have an inactivity threshold (or *time-out*),  $t_1$ . Any user who has been idle for more than  $t_1$  seconds can potentially be disconnected. The difference between different policies is in *which* user (or users) actually do get disconnected. Thus, the number of busy signals is similar across different policies that all have the same value of  $t_1$ . (The numbers are not identical because the set of users that can be potentially disconnected depends on which users were disconnected in the past.) Therefore, the number of busy signals is a good indicator of the overall quality of service, but it is not a good differentiator of the various disconnection policies.

Our other performance metrics are identical to those used in the work of Douglass and Killian. These metrics attempt to measure user “inconvenience” due to disconnections. The approach consists of setting a threshold  $t_2$ . If a user is disconnected and does not become active again within  $t_2$  seconds, then the disconnection is deemed successful and is termed a *soft fault*. If, however, the user becomes active within  $t_2$  seconds, the disconnection is considered a *hard fault* (or just *fault*). (Hard faults were called “bumps” in the Douglass and Killian study, a term borrowed from the disk spin-down domain [DKB95].) The *severity* of a hard fault is a linear measure of how close the user's idle time after disconnection was to the threshold  $t_2$ . (That is, severity is a linear function of idle time after disconnection, such that an idle time  $t_2$  incurs severity 0, and an idle time 0 incurs severity 1.) Two metrics that we used for evaluating disconnection policies are the number of hard faults and the severity of hard faults. Nevertheless, the results using these two metrics were very similar, with the fault severity slightly accentuating the differences between disconnection policies. Since the two metrics yield very similar results, as well as due to lack of space, we will only use the number of hard faults as a metric in the following

section.

The reason for considering only a few of the faults to be significant has to do with the perceived inconvenience of disconnections by the user. A user is wrongly disconnected if her network connection is idle but she is actively using the machine and expects to be connected (e.g., the user may be writing an email reply off-line, which she will later send). In this case, the user will notice the disconnection and reconnect explicitly. This should be counted as a fault by the system because the user was inconvenienced and the lack of network connectivity was immediately noticed. On the other hand, if a user is idle and stays idle after a disconnection, the user was probably truly inactive. In this case, the inconvenience of reconnecting is much less and the user is likely to consider the disconnection justified.

Distinguishing between hard and soft faults may seem strange to readers used to replacement problems. We believe, however, that this metric is truly appropriate for the domain. Additionally, none of the results we present change qualitatively if we consider the total number of faults as a metric. That is, if disconnection policy A is better than disconnection policy B using the number of hard faults as a metric, then A is also better than B using the number of total faults as a metric. What changes is the quantification of how much better A is.

## 5 Experimental Results

We performed several experiments with different disconnection policies, which we outline below. For all the experiments shown here, we set threshold  $t_1$  to be equal to  $t_2$ . This means that a user can be disconnected only if idle for more than  $t_1$  seconds, and the disconnection will be a hard fault if the user's total idle time is not at least  $2t_1$  seconds. The results obtained with  $t_1 = t_2$  are fully representative of all results we have observed for different values of  $t_1$  and  $t_2$  (i.e., we observed no systematic deviation when we experimented with  $t_1 \neq t_2$ ).

For the CIRG algorithm, we used three lists of past idle times (i.e., the value of the parameter  $k$  of Section 3.2 is 3) and the predicted idle time was the average of the predicted time using the three lists. We also performed experiments with one and five lists and the results were very similar.

### 5.1 Replacement vs. Fixed Timeouts

It is, in a sense, unfair to compare a replacement policy with a policy that proactively disconnects users even when the modem pool is not fully utilized. A replacement policy is always going to inconvenience fewer users



while incurring the same number of busy signals. It is, however, interesting to quantify how much better the replacement approach is in a practical setting. Since fixed-idle-threshold disconnection policies are widespread, we show the results of a single experiment below, just as a point of reference. We will not, however, insist on such comparisons any further in the rest of the paper. If keeping users connected has no cost, the replacement approach clearly results in much less inconvenience for users.

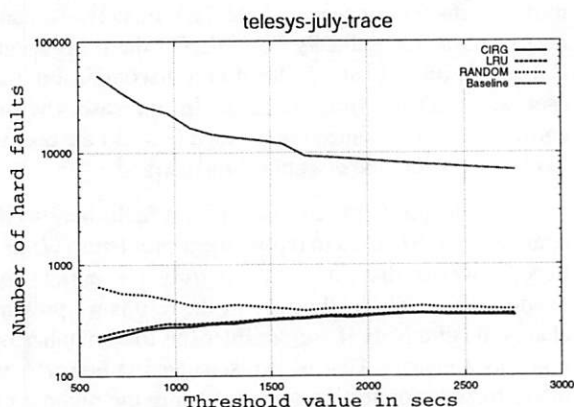


Figure 3: Log plot of hard faults for three replacement policies and a fixed idle threshold policy.

Figure 3 shows the numbers of hard faults incurred when simulating a fixed-idle-threshold disconnection policy and three replacement policies for our first Telesys trace. The simulated modem pool has 1,936 modems (this number is 90% of the maximum number of users who are connected simultaneously in this trace). The threshold value,  $t_1 (= t_2)$ , varies from 600 to 2700 seconds (10 to 45 minutes). (Recall that  $t_1$  is the minimum idle time before a user becomes eligible for disconnection.) For all values of the threshold, the replacement algorithms incur at least one order of magnitude fewer hard faults.

In examining Figure 3 (as well as figures that follow) it may seem awkward that the number of hard faults can rise for higher thresholds. Recall, however, that the figures shown are produced for  $t_1 = t_2$ . When the threshold  $t_1$  increases,  $t_2$  also increases, thus causing many soft faults to be considered hard faults.

## 5.2 Comparison of Replacement Algorithms

A more interesting experiment concerns the comparison of different replacement algorithms. The first question to be answered is whether a simple replacement algorithm, like RANDOM, is good enough. RANDOM

corresponds to the simple idea of replacing any user who has been idle for more than  $t_1$  seconds. The point of reference for the comparison is the LRU algorithm—a common benchmark in replacement problems. LRU replaces the user who has been idle the longest (as long as this is more than  $t_1$  seconds, in our case). The LRU algorithm has been used in replacement settings ranging widely (e.g., from virtual memory to web caching [ASA<sup>+</sup>95]). The next step is to see whether a specialized algorithm can perform better than LRU. As we will see, the CIRG algorithm meets this test.

### Fixed Threshold, Variable Modem Pool Size Results.

Figure 4 shows the results of simulations for all three replacement policies and a wide range of modem pool sizes. The value for threshold  $t_1$  (and, consequently,  $t_2$ ) is set to 600 seconds (10 minutes). We will later examine how our results change under different threshold values.

The ranges of modem pool sizes that we examine contain all reasonable values for practical applications, given each workload. That is, for the low end of the studied range, the workload incurs too many busy signals or too much user inconvenience due to disconnections. For the high end of the studied range, the workload incurs practically no hard faults or busy signals. For the Telesys traces, the chosen range begins at around 70% of the maximum number of simultaneously connected users in the trace.

As can be seen from Figure 4, CIRG and LRU are significantly better than RANDOM in terms of hard faults incurred. The difference ranges from a few tens of percent to over 1000%. Even more importantly, the relative difference is very significant for large modem pools, which are the ones that are going to be encountered in practice. For instance, it would be quite realistic to handle the workload of the July Telesys trace with around 1,900 modems (this setup would suffer around 15 busy signals in 10 days of use). For 1,906 modems, however, LRU incurs 370 hard faults, CIRG incurs 341, and RANDOM incurs 926 faults. Thus, in terms of user inconvenience, RANDOM may not be a good choice for a practical setup. Additionally, we can see that CIRG outperforms LRU. LRU systematically incurs 5% or more, and sometimes over 20% more faults than CIRG.

The second observation we can make in Figure 4 concerns the overall quality of service. We see clearly how a disconnection policy helps maintain a low busy-signal count with fewer modems. Combined with a replacement algorithm that yields a low number of hard faults, the result is a policy that guarantees good service under heavy demand. For instance, the July Telesys workload can be handled using CIRG with 1,875 modems while incurring only 100 busy signals and 500 hard faults dur-



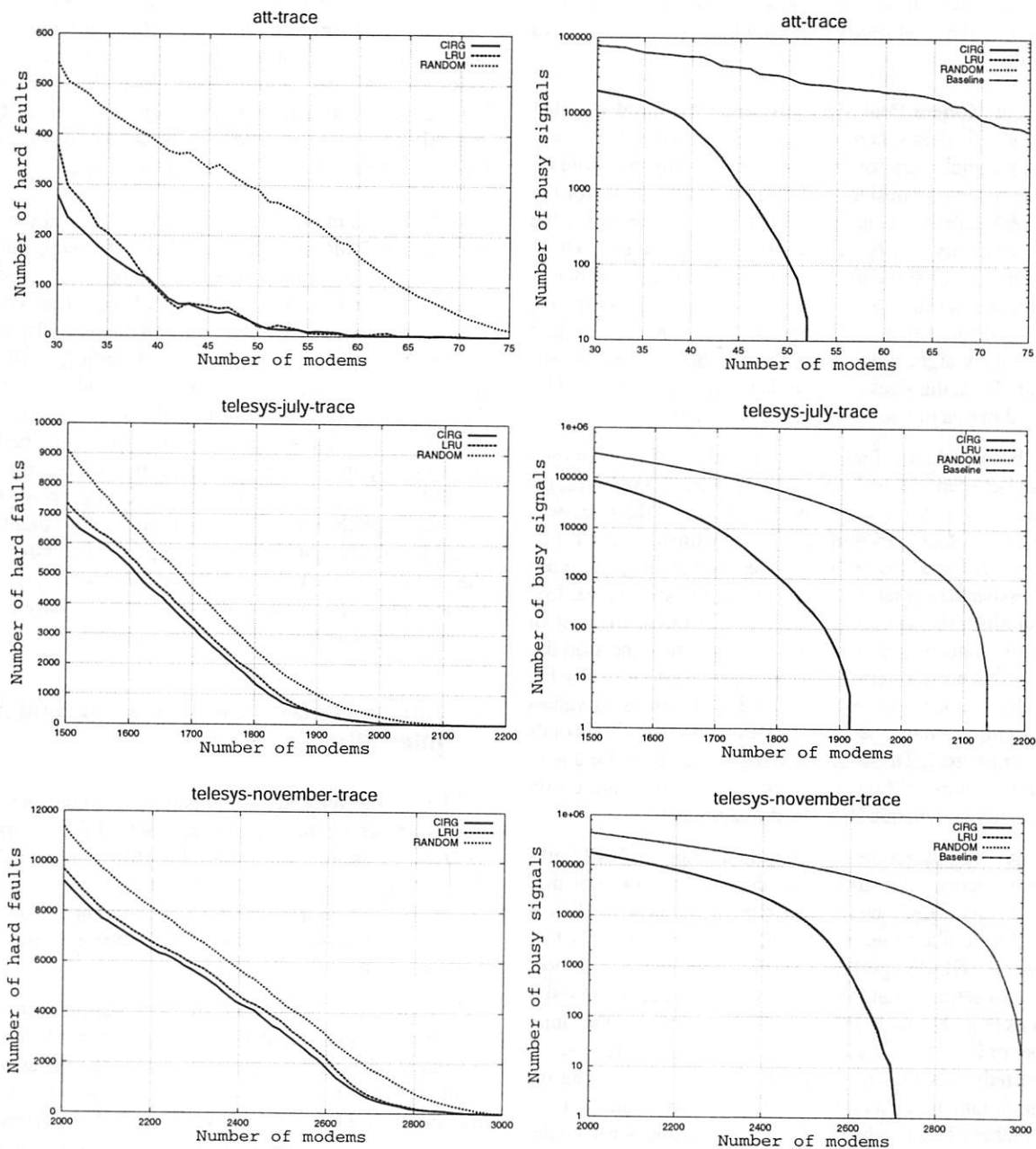


Figure 4: Plots of hard faults (linear scale) and busy signals (log scale) for all traces and a variable number of modems. The value for threshold  $t_1$  is 600 seconds (10 minutes) and  $t_2 = t_1$ . The baseline in the busy signal plots is the number of busy signals incurred with no disconnection policy in place (for the AT&T trace, no user would get explicitly disconnected so this baseline is hardly meaningful). For the busy signal plots, often all three curves for RANDOM, LRU, and CIRG coincide and cannot be distinguished.

ing the course of 10 days (recall that this trace contains over 315,000 connections and user-initiated disconnections). This is 13% fewer modems than would be needed if the system did not disconnect idle users. For comparison, RANDOM needs 1,957 modems to reach 500 hard faults.

**Fixed Modem Pool Size, Variable Threshold Results.** Figure 5 shows how the numbers of hard faults and busy signals vary for different values of the threshold  $t_1$ . The modem pool sizes simulated are 52 modems for the AT&T Labs trace, and 1,936 and 2,721 modems for the Telesys July and November traces, respectively. These modem pool sizes are 90% of the maximum number of modems needed simultaneously for the Telesys traces. For all three traces, the modem pool size is such that few busy signals are incurred for a 600 second threshold. Thus, the sizes are such that they could very well be used in practice to handle these workloads.

For the AT&T Labs trace, the values of  $t_1$  examined are between 300 and 900 seconds (5 and 15 minutes, respectively). Since the environment where the trace was collected had a 15 minute inactivity timeout, these values seem reasonable. The values are certainly more aggressive than what would be expected from a typical ISP, but since the setup uses static IP addresses, the cost of a disconnection is small (i.e., only the reconnection delay and not the termination of open sessions). For the Telesys traces, we experimented with threshold values ranging from 600 seconds (10 minutes) to 2700 seconds (45 minutes). These are more realistic settings for a general purpose ISP where disconnections are more costly due to the dynamic IP address assignment.

As we observe in Figure 5, the number of hard faults may increase for higher threshold values, but not dramatically. LRU and CIRC remain the best predictors of future idle time, with CIRC performing slightly better. RANDOM performs significantly worse than both for all settings that incur a tolerable number of busy signals (e.g., below 1,000 for the Telesys traces). The number of busy signals incurred by all policies increases, expectedly, for higher threshold values. By increasing the minimum inactivity threshold for disconnections ( $t_1$ ), the number of users who can potentially be disconnected decreases rapidly (as seen in the histograms of Figure 2).

We can see from Figure 5 that with 10% fewer modems than the maximum needed simultaneously, we can get a low number of busy signals and hard faults, for high threshold values—over half an hour for the July Telesys trace. The November Telesys trace has worse locality and quickly incurs many busy signals for threshold values above 20 minutes. This is to be expected: users of Telesys who stay idle long, are likely to have ded-

icated phone lines for modem connections. Nevertheless, the increase of Telesys usage between the Summer and Fall trace is mainly due to students. The percentage of students who can afford dedicated phone lines is lower than the corresponding percentage of faculty and staff. Thus, it is natural to have proportionally fewer users who are idle for long in the November trace. We believe that the July trace is more representative of typical ISP subscribers' behavior than the November trace, but have taken no steps towards verifying this.

**Soft Faults** Finally, we present in Figure 6 the numbers of soft faults incurred by all the studied policies. The first column presents the soft faults under constant threshold  $t_1$  and variable modem pool size. The second column presents the numbers of soft faults under constant modem pool size and variable threshold. All parameters are the same as for the corresponding plots in Figures 4 and 5. As we explained earlier, the number of soft faults is not an accurate indication of the performance of a policy—soft faults represent correct predictions that lead to successful disconnections. Nevertheless, we show the soft fault measurements for completeness. The reader can refer to these plots to confirm that the predictions made by CIRC and LRU were very often accurate—the number of soft faults is usually many times higher than the number of hard faults.

## 6 Implementation Considerations and Applicability

The idea advocated in this paper is to consider disconnecting users only if a modem pool is fully occupied. Following this approach, we studied different algorithms for picking users to disconnect. Unfortunately, neither modem servers<sup>7</sup> nor modems have native support for the policies we describe. Fortunately, however, the approach is very easy to implement.

The most straightforward implementation would be one that does not strictly perform replacement but keeps a fixed number of modems unoccupied, as long as users with idle time more than  $t_1$  exist. That is, for a small number  $n$ , if the number of available modems drops below  $n$  and there are users idle for more than  $t_1$  seconds, the system will disconnect one of these users and repeat the process. If there are no users idle for more than  $t_1$  seconds, then all modems can be occupied and other connections will get a busy signal. This implementation

<sup>7</sup>There is no established term for the devices that manage and implement Internet protocols over multiple serial ports. Depending on the exact functionality and marketing decisions of each maker, these are called *modem servers*, *remote access servers*, *terminal servers*, or *communications servers*.

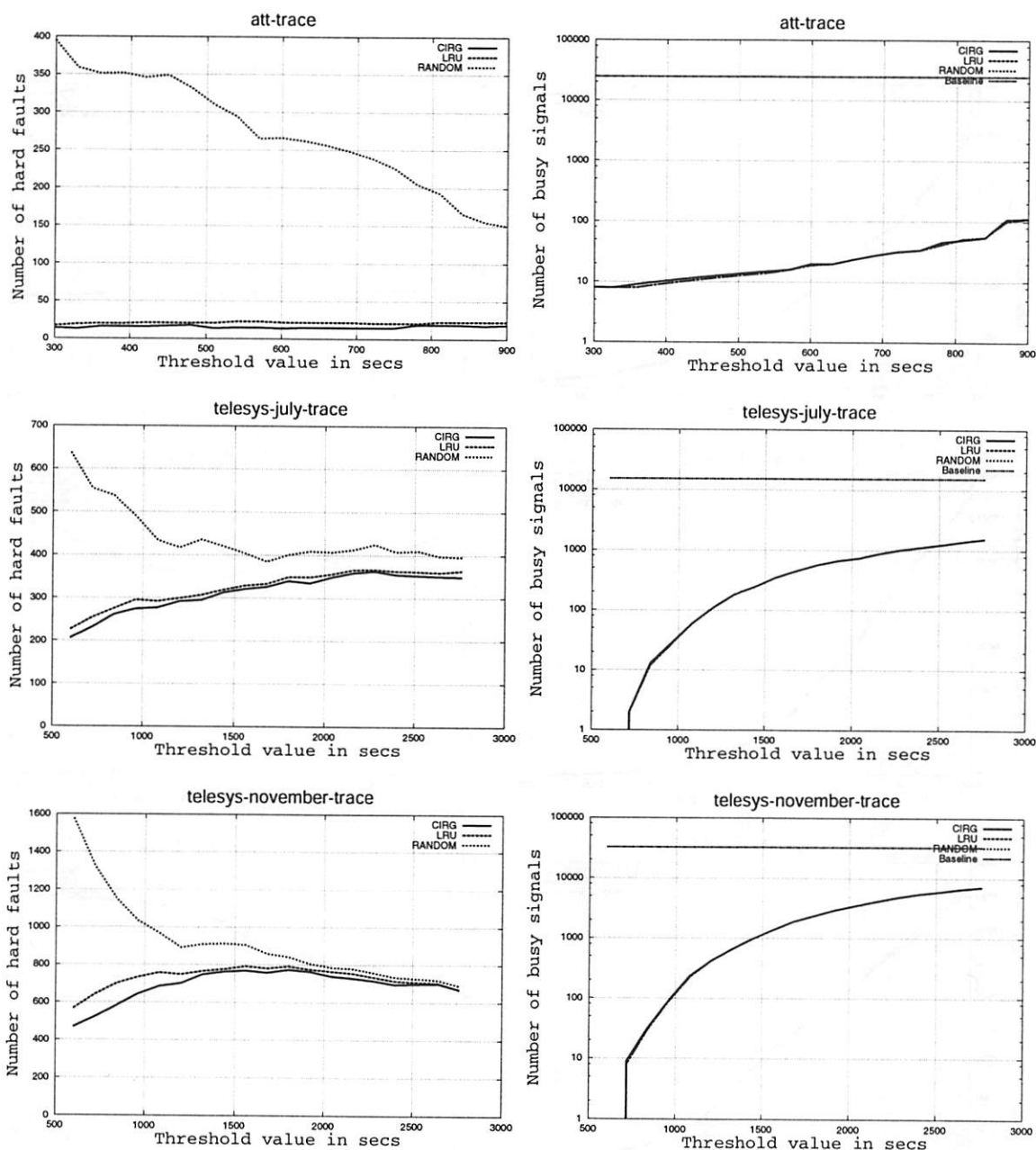


Figure 5: Plots of hard faults (linear scale) and busy signals (log scale) for all traces under variable threshold  $t_1$  (and  $t_2 = t_1$ ). The number of modems for each workload is fixed to a value that yields few busy signals for a 10 minute threshold (for the Telesys workloads, this is 90% of the maximum number of simultaneously connected users in the trace). The baseline in the busy signal plots is the number of busy signals incurred with no disconnection policy in place. For the busy signal plots, often all three curves for RANDOM, LRU, and CIRG coincide and cannot be distinguished.

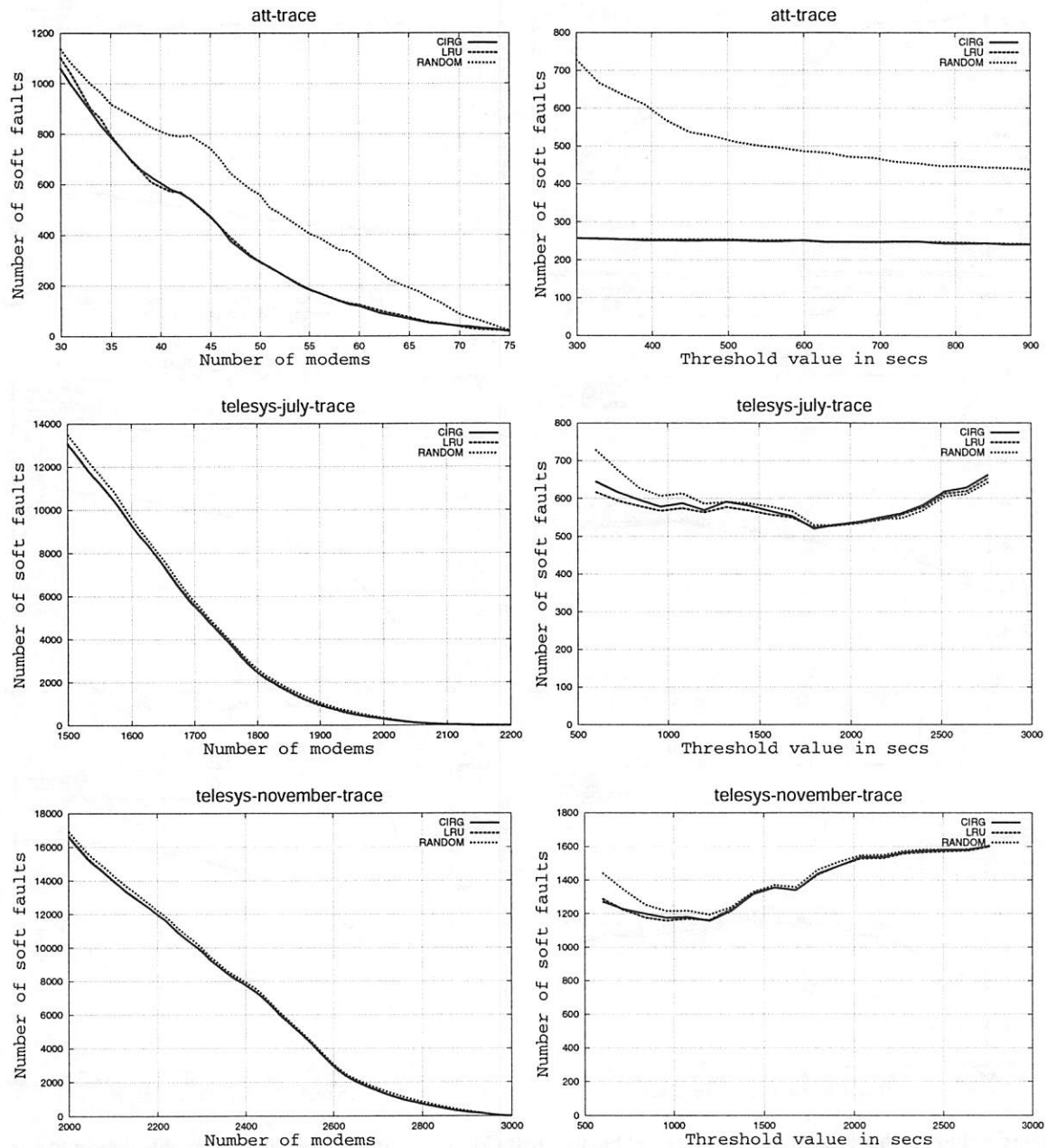


Figure 6: Plots of soft faults for all traces under variable number of modems (first column) and variable threshold (second column). The parameters not shown on the plots are the same as those used in Figure 4 for the first column and in Figure 5 for the second column. The LRU and CIRG curves often coincide and cannot be distinguished.



works well because it does not penalize the system in cases of heavy load that cannot be alleviated with disconnections (all modems can be used). In case of a load that could be lightened with disconnections, it only penalizes the system by keeping  $n$  modems available. The number  $n$  could be quite small relative to the pool size. For the Telesys pool of over 3,000 modems, a value of  $n = 20$  would be reasonable (see below for the rate of connections and disconnections in the Telesys traces).

Additionally, what makes a sophisticated modem disconnection policy easy to implement is that the data and decisions involved are not significant for modern machines. For the Telesys modem pool (which is among the largest unified pools encountered in practice) one has to manage up to a few thousands of modems at any time and a total number of users in the low tens of thousands. Handling replacement policy data structures with this many entries is a simple matter. Even for CIRC and LRU, the more “costly” policies among the ones we studied, updating the data structures and selecting a user to disconnect was, at most, a matter of milliseconds. The total memory required was less than 2MB for CIRC and less than 100KB for LRU at any time. Furthermore, the input data change at human-time rates. Typically, 5 to 10 connections or disconnections per minute were observed in the Telesys trace. As we saw in our experiments, our polling interval of 2 minutes was sufficient for obtaining data such that accurate predictions can be made.

In fact, the problem is computationally simple enough that even a centralized remote implementation is sufficient. (This is certainly not the only option but we discuss it here because of its simplicity.) That is, a remote workstation can be periodically polling all the terminal servers and sending messages that will initiate user disconnections. Many modern communications servers support SNMP (see [CFSD90] and [MR91] for the protocol and the relevant MIB entries), so both the polling and the disconnection commands can be sent remotely over the Internet. Alternatively, a centralized implementation with small proxies that will perform the disconnections at every server seems to be a simple option.

To see how feasible this is, during our trace collection, we polled the over 100 Telesys terminal servers remotely over the Internet using the “finger” command (which uses the *Finger user information protocol* [Zim91]). This method is clearly inefficient because the protocol is not optimized for periodic polling and because the information we needed was less than 5% of the total transmitted data. Nevertheless, our polling took around 50 seconds when done serially and around 15 seconds when done with one process per terminal server (the vast majority of processes finished within 3 seconds but a couple took

longer). Although we have no way of analyzing the delay, it is reasonable to assume that it is primarily due to delay of processing at the terminal server and secondarily due to network latency. The former can be minimized with a less inefficient polling protocol. The latter could be reduced if our machine storing the trace was at closer network proximity of the servers. Nevertheless, even the 15 seconds taken for a remote, inefficient poll are perfectly acceptable—user statistics will not have changed significantly in this time.

## 7 Proactive Disconnections

The main idea explored in this paper is that of performing user replacement in modem pools. The general problem we are addressing, however, is that of predicting future idle times for user connections to the Internet. Thus, our mechanisms could find application in other domains. A prominent opportunity appears in the case of proactive user disconnections. This is the problem studied by Douglass and Killian [DK99]. Their adaptive timeout technique aims at reducing the total connect time across users. Proactive disconnects are interesting when either the service provider or the user is charged more for longer connection times. (As Douglass and Killian admit, “Examples of this in the general ISP market are rare, but some services that function effectively as ISPs do observe this property.”) The tradeoffs involved in such a setting are interesting. For instance, there may be a connection initiation fee for users, so a disconnection should be performed only if the charge of staying connected is higher than the charge of connecting anew.

The formulation of the proactive disconnection problem is similar to that of the replacement problem. There are two thresholds,  $t_1$  and  $t_2$ .  $t_1$  is the minimum idle time before a user is eligible for disconnection.  $t_2$  is the minimum user inactivity time after an automatic disconnection, for the disconnection to be considered successful. (“Successful” may mean cost-effective, psychologically acceptable, or both.)

It is interesting to consider a proactive disconnection algorithm based on the same predictions of future idle time as those made by our replacement algorithms. We have implemented an algorithm called *CIRC-proactive* that uses the same information as the CIRC replacement policy to estimate the future idle time for every user and disconnects any user with an expected future idle time that is longer than  $t_2$ . For instance, assume a value of 5 minutes for  $t_2$ , and a user who has been idle for 6 minutes and has an *idle* list:

2, 5, 13, ...

(these values in minutes). The user will be disconnected,

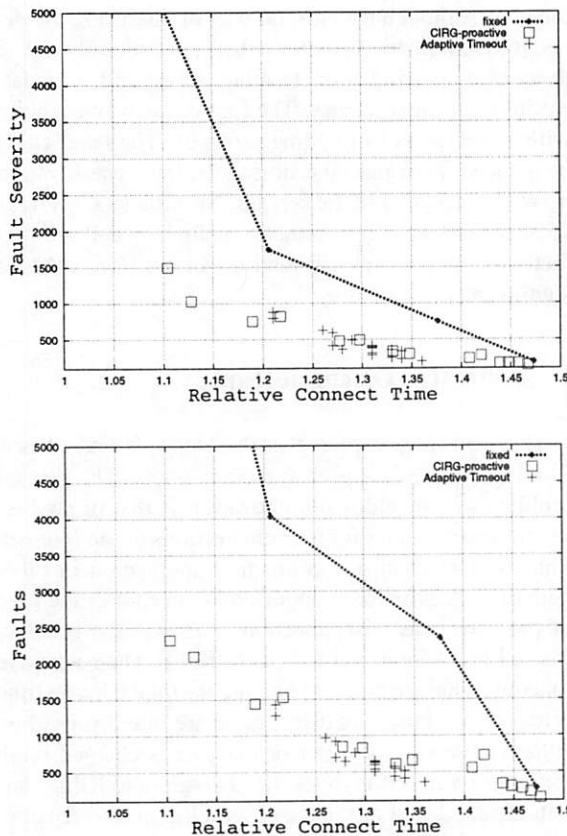


Figure 7: Fault severity and absolute number of faults vs. relative connect time for three policies: a fixed timeout policy, CIRG-proactive, and the Douglin and Killian technique ("adaptive timeout"). Connection times are normalized with respect to the connection time of an optimal, off-line policy.

since her expected future idle time is 7 (for a total idle time of 13).

We have performed experiments with CIRG-proactive and compared it to the Douglin and Killian technique. We should emphasize that our results are preliminary. The reason is that we have not re-implemented the Douglin and Killian mechanism. Hence, we could not examine its effects on our long traces (the Telesys traces). Instead, we use the measurements already computed by Douglin and Killian for the AT&T Labs trace and compare them to the performance of CIRG-proactive.

Figure 7 presents the results of our comparison. It shows two scatter plots: one of fault severities and connection times and another of absolute fault counts and connection times. Both show results for a fixed timeout policy, CIRG-proactive, and the Douglin and Kil-

lian technique applied to the AT&T Labs trace, with 15 workaholics excluded. Connection times are normalized with respect to the connection time of an optimal, off-line (i.e., clairvoyant) policy. That is, the optimal policy would encounter 0 faults for a relative connect time of 1. The value of  $t_2$  is 5 minutes and the values of  $t_1$  for the fixed timeout policy are 2, 5, 10, and 15 minutes. (These values are chosen to be identical with those used by Douglin and Killian in their study.) The values of  $t_1$  for CIRG-proactive are all integral minute values from 1 to 15 minutes. 18 data points for the Douglin and Killian technique are plotted, each for a different combination of adaptivity (multiplicative or additive) and different parameters. As we are comparing to the approach as a whole, we do not distinguish between the different flavors of the Douglin and Killian technique (for this, the reader is referred to [DK99]).

We can see from Figure 7 that CIRG-proactive performs on average just as well as the Douglin and Killian policy and they both perform significantly better than fixed timeouts. Nevertheless, the values for the Douglin and Killian approach are clustered together in the connect-time/user-inconvenience space, while CIRG-proactive offers a wider range of options in the tradeoff between user inconvenience and total connect time. It is worth noting that CIRG-proactive offers this wide range of options with only a single parameter that can be tuned (the  $t_1$  minimum idle time for disconnection) while the Douglin and Killian approach has several degrees of variability (additive vs. multiplicative adaptivity with two numeric parameters for each variant). The wide range of CIRG-proactive values means that we can argue that CIRG-proactive is strictly better than a fixed timeout policy: For each fixed timeout point in the plot, we can find a CIRG-proactive point that is below it and to its left. That is, for each fixed timeout setting, we can find a value of  $t_1$  for CIRG-proactive so that it incurs both a lower fault severity (or fewer faults) and a lower total connect time. The Douglin and Killian technique provides no such guarantee for the values shown.

## 8 Conclusions

In this paper, we examined a disconnection policy for modem pools that performs replacements: users are disconnected only if not enough modems are available for other users to connect. Our idea is certainly not revolutionary but it is nicely evolutionary. If any disconnection policy based on inactivity is to be used, a replacement scheme works best and is easy to apply in practical settings. In our study, we examined several replacement algorithms, and compared their performance in terms of busy signals and user inconvenience.

Perhaps more importantly, however, this paper studied the regularities that arise in Internet users' inactivity times. Based on our analysis, we proposed the CIRG algorithm and showed that it is a very good predictor of future idle times. Our hope is that the main elements of the CIRG approach will also find applications in other domains.

## Acknowledgments

We would like to thank Fred Douglass and AT&T Labs for supplying us with the first of the traces used in our study. We would also like to thank the anonymous referees for their extensive comments. Finally, Don Batory's careful reading helped improve the paper.

## References

- [Aca98] Academic Computing and Instructional Technology Services. Telesys update, September 1998. <http://www.utexas.edu/cc/newsletter/sep98/telesys.html>.
- [Ano97] Anonymous. Steamed customers sue AOL for busy signals. *Information Week*, January 15, 1997.
- [ASA<sup>+</sup>95] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Stephen Williams, and Edward A. Fox. Caching proxies: Limitations and potentials. In *Fourth International WWW Conference*, 1995.
- [BI94] Daniel Barbará and Tomasz Imieliński. Sleepers and workaholics: Caching strategies in mobile environments. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–12, 1994.
- [CFSD90] J. D. Case, M. Fedor, M. L. Schoffstall, and C. Davin. RFC 1157: A simple network management protocol (SNMP), May 1990.
- [DK99] Fred Douglass and Tom Killian. Adaptive modem connection lifetimes. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 27–41, Monterey, California, June 1999. USENIX Association.
- [DKB95] Fred Douglass, P. Krishnan, and Brian Bershad. Adaptive spin-down policies for mobile computers. *Computing Systems*, 8(4):381–413, 1995.
- [GC97] Gideon Glass and Pei Cao. Adaptive page replacement based on memory reference behavior. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1997.
- [MR91] K. McCloghrie and M. T. Rose. RFC 1213: Management information base for network management of TCP/IP-based internets: MIB-II, March 1991.
- [Pha95] Vidyadhar Phalke. *Modeling and Managing Program References in a Memory Hierarchy*. PhD thesis, Rutgers University, 1995.
- [SKW99] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. EELRU: Simple and effective adaptive page replacement. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1999.
- [Zim91] D. Zimmerman. RFC 1288: The Finger user information protocol, December 1991.





# Auto-diagnosis of field problems in an appliance operating system

Gaurav Banga  
Network Appliance Inc.,  
495 E. Java Drive, Sunnyvale, CA 94089  
gaurav@netapp.com

## Abstract

The use of network appliances, i.e., computer systems specialized to perform a single function, is becoming increasingly widespread. Network appliances have many advantages over traditional general-purpose systems such as higher performance/cost metrics, easier configuration and lower costs of management.

Unfortunately, while the complexity of configuration and management of network appliances in normal usage is much lower than that of general-purpose systems, this is not always true in problem situations. The debugging of configuration and performance problems with appliance computers is a task similar to the debugging of such problems with general-purpose systems, and requires substantial expertise.

This paper examines the issues of *appliance-like* management and performance debugging. We present a number of techniques that enable *appliance-like* problem diagnosis. These include *continuous monitoring* for abnormal conditions, diagnosis of configuration problems of network protocols via *protocol augmentation*, path-based problem isolation via *cross-layer analysis*, and *automatic configuration change tracking*. We also describe the use of these techniques in a problem auto-diagnosis subsystem that we have built for the Data ONTAP operating system. Our experience with this system indicates a significant reduction in the cost of problem debugging and a much simpler user experience.

## 1 Introduction

The use of network appliances, i.e., computers specialized to perform a single function, is becoming increasingly widespread. Examples of such appliances are file servers [24, 6], e-mail servers [19, 13], web proxies [25, 5], web accelerators [25, 5, 16] and load balancers [4, 12]. Appliance computers have many potential advantages over traditional general-purpose systems, such as higher performance/cost metrics, simpler configuration and lower costs of management. With the recent growth in the use of networked systems by the non-expert, mainstream population, all of these advantages

have significant importance.

A network appliance is typically constructed using off-the-shelf hardware components. The appliance's service is implemented by custom software running on top of a specialized operating system. (Often the server software is tightly integrated with the OS in the same address space.) The operating system itself is either designed and constructed from scratch, e.g., Network Appliance's Data ONTAP [26], or is a stripped-down version of a general-purpose operating system, e.g., BSDI's Embedded BSD/OS [8].

While appliance computer systems have delivered the promise of higher performance/cost vis-a-vis general-purpose systems, the same is not strictly true of their manageability aspects. While the complexity of configuration and management of appliance computers in normal circumstances is significantly lower than that of general-purpose systems, the debugging of configuration and performance problems of appliances (when they do occur) remains a task that requires substantial operating system and networking expertise. In this respect, appliance systems are similar to general-purpose systems.

This state of technology is not very surprising: Today, the term "appliance-like" is usually taken to mean *specialized to do a single coherent task well*. Specialization of this form has allowed appliance vendors to build and maintain smaller amounts of code than used on general-purpose computer systems. The narrower functionality of appliances has enabled simpler configuration, and more aggressive optimizations leading to superior performance. The ability to easily debug configuration and performance problems has however been a secondary issue so far, and has not received much attention.

Appliance operating systems often contain significant code derived from general-purpose operating systems, particularly UNIX. For instance, the BSD TCP/IP protocol code [33] is a common building block in appliance operating systems. Like general-purpose systems, appliance operating systems export a set of command interfaces that allow users to display values of various statistic counters corresponding to the various events that have

occurred during the operation of the system. Some command interfaces display system configuration parameters. As with general-purpose systems, these command interfaces are the key tools to debugging performance and configuration problems with appliance systems.

For example, the TCP/IP code of many appliance systems exports its event statistics and configuration via a variant of the UNIX *netstat* command. When a person debugging a configuration or performance problem suspects a bug or problem in the network subsystem of the target appliance, she executes the *netstat* command (possibly multiple times with different options) and analyzes the output for aberrations from expected normal values. Any deviations of these statistics from the norm provide clues to what might be wrong with the system. Using these clues, the person debugging the problem may perform additional observations of the system's statistics, using other commands, followed by further analysis and corrective actions (such as configuration changes).

The fundamental problem with this style of statistic-inspection based problem diagnosis is the need for human intervention, and specialized networking and performance debugging expertise in the intervening human. For example, consider a workstation that is experiencing poor NFS [27] file access performance. Assume that the cause of this problem is excessive packet loss in the network path between the client and an NFS server due to a Ethernet duplex mismatch at the server. To diagnose this problem today, the person debugging the problem needs to first isolate the problem to the problematic server, then check the packet drop statistics for the transport protocol in use (UDP or TCP), and correlate these statistics with excessive values for CRC errors or late-collisions maintained by the appropriate network interface device driver<sup>1</sup>. After this, the problem debugger has to check the appropriate switch's configuration to verify the existence of a duplex mismatch.

For any organization engaged in selling and supporting appliance computer systems, it is very expensive to provide a large number of human experts with this level of expertise for the on-site debugging of customer problems. In the absence of sufficient numbers of human experts, problem FAQs, and semi-interactive troubleshooting guides are commonly used by customers and by the (mostly) non-expert customer support staff of the appliance vendors for diagnosing field problems.

Another limitation of this style of problem debugging is that field problems are usually detected *after* they occur. Problems are first detected by unusual behavior (e.g., poor performance) at the application level and then traced back to the cause by a human expert via an ex-

<sup>1</sup> Note that the duplex mismatch cannot be simply avoided as a configuration or installation time automatic check by the server's OS; the Ethernet protocol specification does not contain sufficient logic for an end-system to detect a duplex mismatch.

haustive search and pattern-match through the system's statistics. While there is usually a well-understood notion of *normal* and *bad* values for the various statistics, there exists no software logic to continuously monitor the statistics, and to catch shifts in their values from normal to bad. Problems (and resulting service outages) which could otherwise be avoided by taking timely corrective actions are not avoided.

For all of these reasons, the use of an appliance system can sometimes be a somewhat frustrating experience for a non-expert customer. The subject of this paper is the problem of enabling simple and easy, i.e., *appliance-like*, debugging of the field problems of appliances. We describe four techniques, *continuous statistic monitoring*, *protocol augmentation*, *cross-layer analysis* and *configuration change tracking*, that we have developed to make the diagnosis of appliance problems easier. We also describe the application of these ideas in an auto-diagnosis subsystem of the Data ONTAP operating system.

Specifically, continuous monitoring involves periodically checking the system's collected operational statistics for potential problems, while actively analyzing and fixing whichever problems it can. Protocol augmentation allows configuration problems with a network protocol to be diagnosed using specially constructed higher-level protocol tests. Cross-layer analysis is a path-based approach [23] for isolating a problem with a multi-layered system to a specific system layer. Automatic configuration change tracking keeps track of changes in the system's configuration making it easier to pinpoint a problem to its cause.

Our discussion in the remainder of the paper is set in the context of an appliance operating system. More specifically, we focus on problems that arise with file server appliance systems built and sold by Network Appliance. However, we believe that most of the ideas that we present are directly applicable to the space of general-purpose operating systems. Indeed, the class of field problems involving general-purpose computer systems is much larger than the class of appliance field problems because of the broader functionality and services offered by general-purpose systems. It is probably just as important (and useful) to provide for easier debugging of field problems with general-purpose systems as it is with appliance systems. Later in this paper, we will briefly outline how our auto-diagnosis techniques can be used in a general-purpose operating system, such as BSD.

The rest of the paper is structured as follows. In the next section, we discuss the nature of common field problems of appliance computer systems. In Section 3, we describe the four techniques that we have developed to diagnose such problems automatically and efficiently. In Section 4, we describe the implementation of the NetApp Auto-diagnosis System (NADS). Section 5 describes our experience with this auto-diagnosis system. Section 6

covers related work. Finally, Section 7 summarizes the paper and offers some directions for future work.

## 2 The nature of field problems with appliance systems

Before getting into the details of what can be done to make the debugging of appliance performance and configuration problems easier, it is important to understand the nature of field problems of appliance systems. In this section, we present an overview of the common causes of field problems of appliances and try to give the reader a sense of why it is hard to debug such problems.

As mentioned earlier, for the purposes of concrete illustration, we use the example of a file server (filer) appliance. A filer provides access to network-attached disk storage to client systems via a variety of distributed file system protocols, such as NFS [27] and CIFS [15]. A useful model is to think of a filer's OS as two high-performance pipes between a system of disks and a system of network interfaces. One pipe allows for data flow from the disks to the network; the other carries the reverse flow. Field problems usually arise when something in the filer or in its environment causes one (or both) of these pipes to perform below expected levels.

The taxonomy of common field problems that we describe below was obtained from a detailed study of the call records of Network Appliance's customer service database. We examined information pertaining to customer cases that were handled in the time period February 1994 through August 1999. From this data it appears that the three most important causes of field problems are system misconfiguration, inadequate system capacity and hardware and software faults. The relative ratio of these three problem types is hard to quantify because a large number of customer cases involve more than one subproblem of each type and because the specific mix has varied from month to month and from year to year. However, between these three problem types, they cover about 98% of all field problems.

### 2.1 Misconfiguration

A leading cause of field problems with network appliances is system misconfiguration. This may seem somewhat paradoxical since by definition an appliance is a simple computer system that has been specially developed to perform a single coherent task. This definition is supposed to allow an appliance system to be simpler to configure and use. In reality, appliances by themselves are usually much simpler than general-purpose systems. However, the task of making appliances work correctly in a real network in a variety of application environments may still have significant configuration complexity.

One major reason for the configuration complexity associated with a appliance system is that an appliance in use is only a part of a potentially complex distributed

system. For example, the perceived performance of a filer is the performance of a distributed system consisting of a client system (usually a general-purpose computer system) connected via a potentially complicated network fabric (switches, routers, cables, patch panels etc.) to the filer. These components typically come from different vendors and need to be all configured and functioning correctly for the filer to function at its rated performance. Unfortunately, this does not always happen for a variety of reasons, as discussed below.

First, the client system usually has a fairly complicated and error-prone configuration procedure. The client's configuration complexity is much more so than the filer's because the client is a general-purpose system. Often, the default configurations in which most client systems ship are simply not set for optimal performance. (This issue of default configuration is discussed in somewhat more detail later.) In many cases, the configuration controls are too coarse for any allowable setting to result in good performance for all activities that the general-purpose client may be engaged in.

Second, while most components of the network fabric are appliances (and therefore presumably easier to configure than client systems), there are numerous potential incompatibilities between them. For example, it is not uncommon for implementations of network communication protocols from different vendors to not work with each other. Usually, the corresponding vendor documentation clearly states this incompatibility, but customers try to use the incompatible implementations anyway, and the result is a field problem.

Perhaps more importantly, some commonly used standard network protocols have serious inadequacies. For example, the Ethernet standard includes an *auto-negotiation* protocol for negotiating the link speeds of the communicating entities. The standard does not provide for reliable negotiation of duplex settings. As a result, perfectly legal configuration settings for link and duplex at two communicating endpoints may result in a duplex-mismatch, a misconfiguration whose effect on a filer's throughput is disastrous.

Furthermore, network components often use protocols that are vendor-specific or ad-hoc standards. These "early" protocols work well in most situations, but not at all (or poorly) in other circumstances. In the fast moving world of network technology, there are a fair number of ad-hoc, unstandardized, or incomplete protocols in wide use at any given time. An example of this is the EtherChannel link aggregation protocol. This protocol does not specify the algorithm for performing load balancing of network traffic between the links of the EtherChannel. Vendors have their own propriety methods for this process, often with surprising interactions with how the client systems and the rest of the network elements are set up. These interactions sometimes have a significant



effect on performance and result in field problems.

A second important cause of the configuration complexity associated with appliance systems is the sub-optimal management of configuration parameters. The appliance philosophy is to expose a very small number of configuration parameters at installation. There is a second tier of parameters that are assigned default values which result in good performance in the majority of installations. For some installations with atypical workloads, these settings may not be optimal. There is usually no automatic logic to tune these second tier parameters. In these cases, these knobs may require tuning by an expert for good performance.

With the widespread increase in the variety and number of appliance users, this atypical population can become a significant overall number, potentially resulting in a large number of field problems. This problem of configuration parameter management also exists with general-purpose operating systems, including systems that are used as clients for filers. In fact, with general-purpose systems, a large number of parameters often need to be tuned for a typical user environment.

## 2.2 Capacity problems

A second class of field problems with appliance systems arise because of their poor handling of capacity overloads<sup>2</sup>. Most commonly-used general-purpose operating systems, and many appliance operating systems, perform well when the request load to which the system is being subjected lies within the capacity of system, but poorly when the offered load exceeds the capacity of the system [20, 7]. Historically, the problem of poor overload performance of computer systems is well known, but has been deemed of somewhat marginal importance. In most circumstances it is not desirable to operate a system under overload conditions for any length of time; instead, the focus so far has been to avoid overload by trying to ensure that there are always sufficient hardware resources available in order to handle the maximum offered load.

In the filer appliance market, systems are often purchased by customers with a certain client load in mind. The number and types of systems purchased is chosen based on rated capacities of the filers, by in-house benchmarking, or from knowledge based on prior-use of filers. Filers are usually assigned rated capacities based on their performance under some standardized benchmark, e.g. the SpecFS (SFS) benchmark [30]. For many customers' sites, however, the request load profile is significantly different from the SFS profile, and the real capacity of a filer in operation may be very different from its rated

<sup>2</sup>We use the term "capacity overload" to refer to a broad class of situations where the server system cannot handle the full client request load that it is subject to because of some system resource that is not available in sufficient quantity. These resources include "soft" system resources such as memory buffers, file system buffers etc.

capacity. When offered load does exceed real capacity, poor performance and a field problem results.

## 2.3 Hardware and software faults

Last but not least, some field problems with appliances occur because of software and hardware faults. Unlike the other causes of field problems discussed above, faults are the result of some bug in the system's implementation, and usually result in system down-time. For a mature system made by a technically sound organization, the number of field disruptions due to faults should be very small.

Field problems due to faults are not discussed further in this paper. The techniques that we describe in this paper to enable easy debugging of field problems may have some applicability to diagnosing certain types of field disruptions due to faults, but in this paper we restrict our focus to diagnosing configuration and capacity problems.

## 2.4 Why are field problems hard to debug?

When a field problem occurs with an appliance system due to any of the reasons described above (except faults), it is often hard to debug. Consider a filer customer who observes performance that is substantially lower than the filer's rated performance. The reason for this poor performance may be a misconfiguration somewhere in the client-to-filer distributed system, i.e., in the client, in the filer, or in the network fabric. Alternately, the problem may be an overloaded filer; this particular environment may have an atypical load and the filer may have a lower capacity for this workload than for the standard SFS workload.

As the end effect of all of these potential causes is usually the same, i.e., poor file access performance as seen from the client system, it is not easy to discern the exact cause of the problem. The problem debugger is forced to perform a sanity check of *all* the components of the client-to-filer distributed system in order to ensure that each component is functioning correctly. For the filer, this implies a verification of all filer subsystems performed by invoking the various statistic commands and analyzing the output for aberrations.

This process is time-consuming, tedious and error-prone. As explained earlier, this task requires a fair amount of expertise, and a certain debugging "instinct" that comes from experience. This task is also complicated by the fact that the person debugging the field problem, being a member of the filer vendor's organization, often has no direct access to the system being debugged. In that case, the various statistic commands are executed by the customer who is in communication with the support person via email or phone. This aspect of the problem debugging process makes it slow, causing large down-time. Combined with the high expectations of *appliance-like* simplicity that most appliance



customers have, it makes the problem debugging experience frustrating for both parties involved, the customer and the support person.

The discussion above is fully applicable to general-purpose systems; appliances are usually considerably easier to debug than general-purpose systems. However, the debugging of field problems with appliances is certainly not as simple, or “appliance-like”, as we would like. In the next section, we will present a new problem diagnosis methodology that attempts to apply the appliance to the debugging of field problems with appliance systems.

### 3 Problem auto-diagnosis methods

In this section, we describe a new methodology that we have developed to make the diagnosis of appliance field problems simpler. Our goal in designing this methodology was to enable problem diagnosis to be as automatic, precise and quick as possible. We wanted to eliminate the need for expert human intervention in the problem diagnosis process whenever possible. Furthermore, for those situations where expert manual analysis is necessary, we wanted to provide powerful debugging tools, precise and comprehensive system configuration (and configuration change) information and the results of partial auto-analysis to the human expert, allowing for fast diagnosis and smaller down-times.

Our problem diagnosis methodology is based on four specific techniques, i.e., continuous monitoring, protocol augmentation, cross-layer analysis and configuration change tracking. Each of these techniques is described in detail below. In this section, we will focus on the fundamental principles underlying these techniques; the next section will contain specific details about the application of these techniques in the auto-diagnosis subsystem of the Data ONTAP operating system.

We will also briefly discuss issues related to the extensibility of our new problem diagnosis methodology. This feature is important for the problem auto-diagnosis system to be maintainable in the field.

#### 3.1 Continuous monitoring

As described in Section 1, current appliance operating systems maintain a large number of statistics. To help in auto-detecting and diagnosing problems, we have developed a method of continuous statistic analysis layered on top of this statistic collection procedure. Software logic in the appliance system continuously monitors the system for problems, actively analyzing and fixing whatever problems it can. Continuous monitoring has two components to it, a passive part and an active part.

The passive part of continuous monitoring is a statistic monitoring subsystem of the appliance’s operating system. This subsystem periodically samples and analyses the statistics being gathered by the operating sys-

tem. It automatically looks for any aberrant values in these statistics and applies a set of predefined rules on any aberrations from expected “normal” values to move the system into one of a set of error states. For example, a filer may continuously monitor the average response time of NFS requests. A capacity overload situation is flagged when the response time exceeds a high-water mark.

Some abnormal system states may correspond uniquely to specific problems; other states may be indicative of one of a set of possible problems. In the latter case, the continuous monitoring subsystem may also automatically execute specially designed tests in order to pin-point the specific problem with the system. This is the active part of continuous monitoring. For example, a large number of packet losses on a TCP connection at a filer may be indicative of, among other problems, a duplex mismatch at one of the filer’s network interfaces or a high level of network congestion in the path from the relevant client to the filer. We can use the techniques described below in sections 3.2–3.4 to differentiate between these problems.

Making continuous system monitoring viable involves the following:

- Development of software logic that formally codifies the informal notion of *expected* statistic value. This activity must be performed for all of the statistics that are gathered by the system. The end-result of this activity is a set of equations that test the state of the system and return either “GOOD” or move the system into an “ERROR” state.
- Development of software logic that selects an appropriate problem pin-pointing procedure when one of several problems is suspected based on observations of aberrant system statistics.
- Development of formal procedures for pin-pointing common field problems of appliances.

Formally codifying the notion of expected values of the various statistics is a hard problem. This is because, in general, the normal values of the various system statistics and the relative sets of values that indicate error conditions depend on how a particular system is being used. For example, an average CPU utilization of 70% might be OK for a system that is usually not subject to bursts of load that greatly exceed the average. This may, however, be a big problem for a system whose peak load often exceeds the average by large factors.

To make the development of this logic tractable, it may be necessary to be somewhat conservative in the choice of the specific problems to be characterized. For any particular appliance, this logic can start from being very simple, codifying only the most obvious problems initially, and move towards more complex checks as the

appliance's vendor gains experience with how the appliance is used in the field. At any point in an appliance's life-cycle, there will be some logic that can be completely automatically executed and its results presented directly to the customer/user. Other, more complex logic may attempt to perform partial-analysis and make these results available to a support person looking at the system, should manual debugging be necessary. Still more complicated analysis may be left to the human expert.

The idea behind developing active tests for pin-pointing problems is to try to mimic the activity of problem analysis by a human expert. While debugging a field problem, this person may take a certain set of statistic values as a clue that the system is suffering from one of a certain set of problems. He may then execute a series of carefully constructed tests to verify his hypothesis and pin-point the exact problem. Continuous monitoring with active tests attempts to mimic this debugging style.

The algorithm development activity for active tests motivates the next three techniques, i.e., protocol augmentation, cross-layer analysis and configuration change monitoring that we describe below. The software logic to trigger these tests is usually straightforward, once the main logic of continuous monitoring is in place.

Of course, the continuous monitoring logic has to be lightweight. It should work with as few system resources as possible and should not impact system performance in any noticeable way. The active component of system monitoring should not affect the system's environment, e.g., the network infrastructure to which it is attached, in any adverse manner. We will discuss some practical aspects related to the user-interface of the continuous monitoring subsystem in the next section.

Once continuous monitoring is in place, it has many benefits. A sizable fraction of field problems can be auto-diagnosed without intervention of the support staff. If expert intervention is needed, all information that is normally gathered by a human expert after (potentially time-consuming) interaction with the customer is already available. Changing system behavior that slowly moves the system towards an ERROR state may be detected early, and corrected, before it results in down-time. For example, increasing average load that slowly drives a system into capacity overload can be auto-detected.

Similarly, other shifts in a system's environment, such as the load mix to which it is subjected, may be auto-detected and suitable action may be initiated. Continuous monitoring may also help an appliance vendor in tuning his product better because he now has access to more detailed information about the various customer environments in which the product operates. In essence, continuous monitoring is like having a dedicated support person attached to every appliance in the installed base, but at a very small fraction of the cost.

### 3.2 Protocol augmentation

The technique of protocol augmentation refers to the process by which a higher-level protocol in a stacked modular system configures and operates a lower-level protocol through a series of carefully chosen configurations and operating loads. The goal of protocol augmentation is to determine the optimal configuration of the lower-level protocol when it is impossible to determine this setting within the protocol itself. This is necessary because the lower-level protocol is either *inadequate*, *incompletely specified* or if one of the communicating entities has a *broken* protocol implementation.

As briefly mentioned in the previous section, some network protocols are *inadequate* in that it is impossible to detect configuration problems of the communicating entities within the protocol itself. An example of this is Ethernet auto-negotiation, which does not always allow for the correct negotiation of the duplex settings of the communicating entities.

Some network protocols are *incompletely specified*. For instance, the algorithms for congestion control were not specified as part of the original TCP protocol standard. Congestion control was incorporated by most TCP implementations much later from a de-facto standard published by the researchers who developed these algorithms. Often, such de-facto standards involve areas of the protocol that are not necessary for correctness, and are therefore not enforced. A TCP implementation that does not perform congestion control correctly may still be able to communicate adequately with other TCP implementations; however, correct congestion control is imperative for system-wide stability and performance.

A number of protocol implementations, especially where unofficial de-facto standards are involved, are *broken*. For example, some commonly used auto-negotiating Gigabit Ethernet devices detect link only if the peer entity is also set to auto-negotiate.

When a problem occurs due to any of the three reasons mentioned above, the continuous monitoring subsystem detects this situation and flags an error condition. If an active test has been associated with the equations that triggered this error state, this active test is executed. The active test will use protocol augmentation to mimic a human expert in the debugging process. For example, a test designed to detect an Ethernet duplex mismatch may try all legal settings of speed and duplex coupled with initiation of carefully constructed Ethernet traffic. It may analyze the resulting change in system behavior to determine the correct settings for speed and duplex.

Protocol augmentation is a powerful technique that can be used as a guiding framework to formalize many ad-hoc problem debugging techniques used by human experts. Any manual debugging technique that involves a series of steps where network configuration changes alternate with functionality or performance tests (to val-

idate the configuration) is really a form of protocol augmentation. Using this technique as a design guide, we can come up with problem diagnosis procedures that are more precise and systematic than the ad-hoc techniques normally used in manual diagnosis. In the next section, we will describe some examples of the use of this technique in designing automatic problem diagnosis tests for commonly occurring filer problems.

### 3.3 Cross-layer analysis

Many subsystems of appliance operating systems are implemented as stacked modules. For example, the TCP/IP subsystem consists of the link layer, the network layer (IP), the transport layer (TCP and UDP) and the application layer organized as a protocol stack. Each layer of a stacked set of modules maintains an independent set of statistics for error conditions and performance metrics. When a problem occurs, it may manifest itself as aberrant statistic values in multiple layers in the system. In classical systems, there is no logic that correlates these aberrant statistic values across different system layers.

Cross-layer analysis is a new technique whereby statistic values in different layers of a subsystem are linked together, and co-analyzed. Essentially, we identify paths [23] in OS subsystems and link together the statistic values in the various layers that each path crosses. When continuous monitoring detects a problem in a path, the various layers of the path can be quickly examined to isolate the specific problem.

As a debugging technique, cross-layer analysis is a formalization of the ad-hoc technique used by human experts in manual problem debugging where an observation of an aberrant statistic value in one layer triggers a study of the statistic values of an adjacent layer. Considering the pipeline analogy of an appliance operating system, cross-layer analysis guides the debugging process by tracing through and ensuring the health of the various layers that implement the disk-to-network pipes.

As a guiding framework, cross-layer analysis can aid the design of logic that causes the continuous monitoring subsystem to trigger the various active tests. For example, the logic to perform a check for a duplex mismatch on a network interface may be triggered by an observation of excessive TCP level packet loss in a connection that goes through this interface. Cross-layer analysis can also guide the design of the statistic data and its collection logic so as to allow problem debugging to be easier. For example, the need to do cross-layer analysis may require a modification of the BSD *tcpstat* and *ipstat* structures so as to keep some statistics on a per flow basis.

### 3.4 Automatic configuration change tracking

Many field problems with appliance systems are caused by changes in the system's environment. These include system configuration changes and changes in the offered load. As described earlier, there is a lot of value

in continuous monitoring of system statistics to notice shifts in metrics like average system load. Likewise, it is useful to track changes in the system's configuration, both explicit as well as implicit.

Automatic tracking of configuration changes is useful in finding the cause of appliance problems that occur after a system has been up and running correctly for some time. This technique also helps in prescribing solutions for the problems found by other auto-diagnosis methods. In many organizations, there are multiple administrators responsible for the IT infrastructure. Configuration change tracking allows for actions of one administrator that result in an appliance problem to be easily reversed by another administrator. This is also useful where administrative boundaries partition the network fabric and the clients from the filer.

The fundamental motivation behind automatic configuration change tracking is to automatically gather information that is asked for by human problem debuggers in a large majority of cases. Anyone familiar with the process of field debugging probably knows that one of the first questions that a customer reporting a problem gets asked by the problem solving expert is: "What has changed recently?" The answer to this is often only loosely accurate (especially in a multi-administrator environment), or even incorrect, depending on the skill level of the customer/user. Automatic configuration change tracking makes precise and comprehensive state change information available to the problem solver, i.e., the auto-diagnosis logic or a human expert.

Configuration changes are tracked by a special module of the appliance OS. As hinted above, configuration changes are of two types: the first type of changes are explicit, and correspond to state changes initiated by its operator. The second type of changes are implicit, e.g., an event of link-status loss and link-status regain when a cable is pulled out and re-inserted into one of a filer's network interface cards. The system logs both explicit and implicit changes. The amount of change information that needs to be kept around is a system design parameter, and may require some experience in getting to optimal for any particular appliance.

Given comprehensive configuration change information, when a problem occurs the various events between the last instance of time which was known to be problem free to the current event are examined and analyzed. The software logic to do this analysis, like the logic for continuous monitoring, is system specific and may need to be evolved over time. In some cases, the auto-diagnosis system can directly infer the cause for the field problem, and report this. In other cases, the set of all applicable configuration changes can be made available to the human debugging the system.

Note that it is not absolutely imperative to log *all* relevant configuration change information. (In fact, some



configuration changes may not be easily visible to the appliance. For example, the path between a client and a filer may involve multiple routers, and it may be possible to change/re-configure one of these without the filer noticing any changes in its environment.) State change information is however only a set of hints that guide the problem diagnosis process and make it easier. If some relevant state change information is not logged, diagnosing the cause of a specific problem may become harder, but not necessarily impossible. In our experience, logging even a modestly-sized, carefully chosen set of configuration change information, is extremely valuable in the problem diagnosis process.

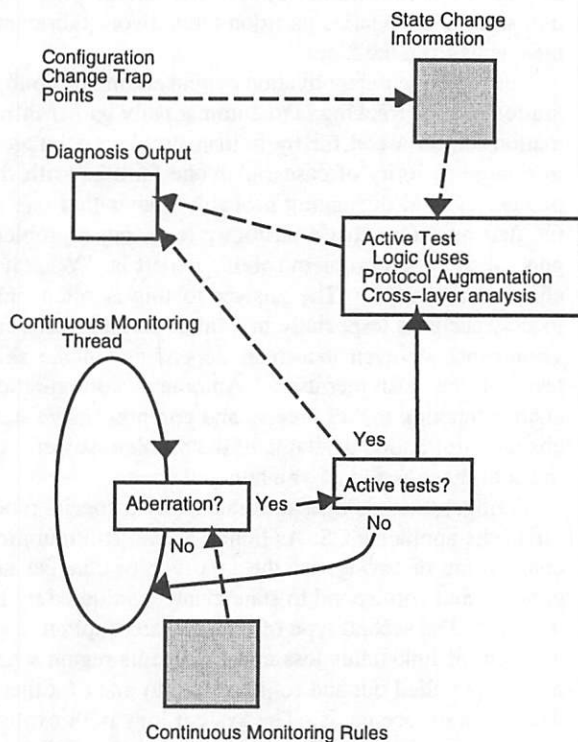


Fig. 1: Role of different auto-diagnosis techniques.

Figure 1 shows the role of the various auto-diagnosis techniques in the problem diagnosis process. In the figure, dashed lines indicate flow of data while solid lines indicate flow of control. The shaded rectangles indicate stores of data or logic rules. The unshaded rectangles indicate processing steps. Note that the problem diagnosis process uses all the techniques we described above. The techniques are complementary and designed to work with each other; they are *not* different types of procedures targeted to address disjoint problem sets.

### 3.5 Extensibility issues

It is important for an auto-diagnosis system built around the techniques described above to be extensible. As explained above, the checks and actions performed by

the continuous monitoring logic need to be developed in a phased and conservative manner. Each time a new version of this logic is available, a vendor may want to upgrade the systems in the field with this logic, even if the customers do not wish to upgrade the rest of the system. A customer may not wish to take on the risk associated with a new software release, or may not want to pay for the release, especially if it does not contain any functionality that the customer needs. It is, however, usually in the vendor's interest to upgrade the auto-diagnosis logic because of the little associated risk and potential benefits of lower support costs.

For example, an appliance problem may have been first discovered at one customer's installation because of an environment change, e.g., the addition of a new model of some hardware in the network fabric. In some cases, significant effort by human experts may be required to debug this problem since it has not been seen before. Ideally, we would like to leverage off this effort by codifying the debugging logic used in this manual diagnosis into the appliance's auto-diagnosis logic and upgrading the auto-diagnosis subsystems of *all* the systems in the field. This may save a lot of time and effort by auto-diagnosing subsequent instances of this problem which would otherwise require significant human intervention.

Extensibility can be achieved in a variety of ways. One method is for the continuous monitoring system to use a configuration file containing equations that define the various periodic checks that the monitoring system performs and conditions that trigger the flagging of an ERROR state, or cause an active subtest to be executed. This requires a language to express the logic of the periodic checks, and an interpreter for this language to be part of the problem auto-diagnosis subsystem.

## 4 Implementation of the NetApp Auto-diagnosis System

We have implemented a semi-automatic problem diagnosis system, the NetApp Auto-diagnosis System (NADS), in the Data ONTAP operating system. This system applies the techniques described in the previous section to field problems with filers and NetCache appliances. Currently this auto-diagnosis system only targets problems related to the networking portion of Data ONTAP, and some of the interactions of this code with the rest of Data ONTAP. Extension of the auto-diagnosis system to other ONTAP subsystems is in progress.

An interesting social problem that we had to address while developing the auto-diagnosis system was to how not to make the auto-diagnosis logic intrusive. We did not want our expert customers to be turned-off by an overbearing problem diagnosis "assistant" and immediately disable the auto-diagnosis system. We also did not want our non-expert customers to be lead off on a side-track by a bug in the auto-diagnosis logic. For this rea-



son, we decided that we would make the auto-diagnosis process semi-automatic initially, and later, as both we and our customers gained experience with the system, make it fully automatic.

#### 4.1 Core implementation

In its current form the NetApp Auto-diagnosis System consists of a continuous monitoring subsystem and a set of diagnostic commands. ONTAP's continuous monitoring logic consists of a thread that wakes up every minute and performs a series of checks on statistics that are maintained by various ONTAP subsystems. These checks may flag the system as being in an ERROR state. This logic is currently hard-coded into ONTAP (as C code tightly integrated into the kernel) and needs to be tuned with every maintenance release. Threshold values and most constants used by this logic are read from a file present root filesystem of the filer<sup>3</sup>. This logic does not yet perform any output for direct user consumption; nor does this logic execute any active tests. Instead this output is logged internally in ONTAP for consumption by the various diagnostic commands, which also execute any active tests that are needed. Since in ONTAP all commands are implemented in the same address space as the kernel, it is straightforward for the data gathered by continuous monitoring to be accessed by the diagnostic commands. Likewise, it is easy for the active test logic to be executed by the diagnostic commands.

When the customer or a support person debugging a field problem suspects that the problem lies in the networking portion of ONTAP, she executes the *netdiag* command. The *netdiag* command analyzes the information logged by the continuous monitoring subsystem, performing any active tests that may be called for and reports the results of this analysis, and some recommendations on how to fix any detected problems, to the user. Our plan is to have the computation of the various diagnostic commands be performed automatically after the next few releases of ONTAP.

The checks that ONTAP's continuous monitoring system performs and the various thresholds used by this logic have been defined using data from a variety of sources of collected knowledge. These include FAQs compiled by the NetApp engineering and customer support organizations over the years, troubleshooting guides compiled by NetApp support, historical data from NetApp's customer call record and engineering bug databases, information from advanced ONTAP system administration and troubleshooting courses that are offered to NetApp's customers, and ideas contributed by some problem debugging experts at NetApp.

The specific monitoring rules and the values of various constants and thresholds used by the monitoring logic and even the list of problems that ONTAP's auto-

diagnosis subsystem will address when complete is fairly extensive; due to space considerations we will not cover this information in full detail. Instead, we will restrict the following discussion to some common networking problems that ONTAP currently attempts to auto-diagnose. We will describe the set of problems targeted by this logic and illustrate its operation with two examples.

At the link layer, ONTAP attempts to diagnose Ethernet duplex and speed mismatches, Gigabit auto-negotiation mismatches, problems due to incorrect setting of store and forward mode on some network interface cards (NICs), link capacity problems, EtherChannel load balancing problems and some common hardware problems. At the IP layer, ONTAP can diagnose common routing problems and problems related to excessive fragmentation. At the transport layer, ONTAP can diagnose common causes of poor TCP performance. At the system level, ONTAP can diagnose problems due to inconsistent information in different configuration files, unavailability or unreachability of important information servers such as DNS and NIS servers, and insufficient system resources for the networking code to function at the load being offered to it.

To see how the techniques described in the previous section are used, consider the link layer diagnosis logic. The continuous monitoring system monitors the different event statistics such as total packets in, total packets out, incoming packets with CRC errors, collisions, late collisions, deferred transmissions etc., that are maintained by the various NIC device drivers. Assume that the continuous monitoring logic notices a large number of CRC errors. Usually, this will also be noticed as poor application-level performance.

Without auto-diagnosis, the manner in which this field problem is handled depends on the skill level and the debugging approach of the person addressing the problem. Some people will simply assume bad hardware and swap the NIC. Other people will first check for a duplex mismatch (if the NIC is an Ethernet NIC) by looking at the duplex settings of the NIC and the corresponding switch port, and if no mismatch is found may try a different cable and a different switch port in succession before swapping the NIC.

With the *netdiag* command, this process is much more formal and precise (Figure 2). The *netdiag* command first executes a *protocol augmentation* based test for detecting if there is a duplex mismatch. Specifically, the command forces some "reverse traffic" from the other machines on the network to the filer using a variety of different mechanisms in turn until one succeeds. These mechanisms include an ICMP echo-request broadcast, layer 2 echo-request broadcast and TCP/UDP traffic to well-known ports for hosts in the ARP cache of the filer. First the ambient rate of packet arrival at the filer using whatever mechanism that generated sufficient return traf-

<sup>3</sup>See the subsection on extensibility for how this is going to change.

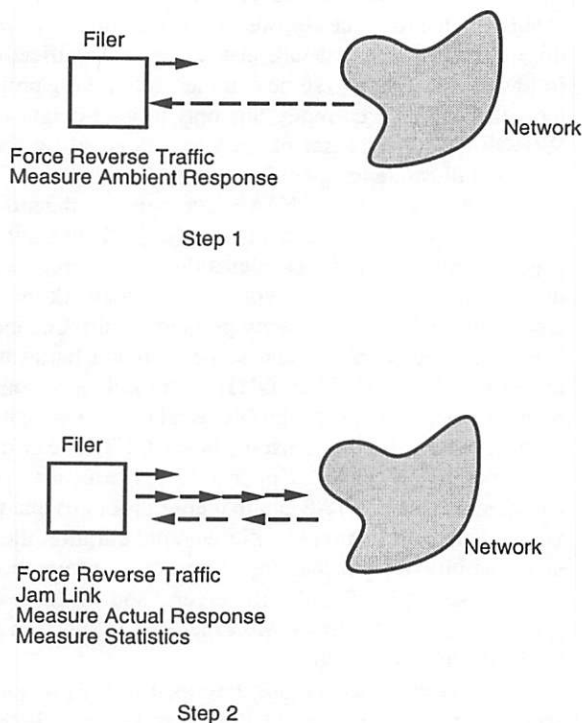


Fig. 2: Diagnosing a duplex mismatch using protocol augmentation.

fic is measured (Figure 2, Step 1). Next this reverse traffic is initiated again using the same mechanism as before and the suspect outgoing link is jammed with back-to-back packets destined to the filer itself (which will be discarded by the switch). The reverse traffic rate is then measured, along with the number of physical level errors during the jam interval (Figure 2, Step 2). If there is indeed a duplex mismatch, these observations are sufficient to discover it, since the reverse rate will interfere with the forward flow inducing certain types of errors only if the duplex settings are not configured correctly. In this case, the *netdiag* command prints information on how to fix the mismatch.

If the reason behind the duplex mismatch is a recent change to the filer's configuration parameters, this information will also be inferred by the auto-diagnosis logic and printed for the benefit of the user. If the NIC in question noticed a link-down-up event in the recent past and no CRC errors had been seen before that event, the *netdiag* command will print out this information as it could indicate a switch port setting change, or a cable change or a switch port change event which might have triggered off the mismatch. This extra information, which is made possible by automatic configuration change tracking, is important because it helps the customer discover the cause of the problem and ensure that it does not re-

peat. This problem may have been caused by, for example, two administrators inadvertently acting at cross-purposes.

If there is no duplex mismatch, the *netdiag* command prints a series of recommendations, such as changing the cable, switch port and the NIC, in the precise order in which they should be tried by the user. The order itself is based on historical data regarding the relative rates of occurrence of these causes.

As another example, consider the TCP auto-diagnosis logic. ONTAP's TCP continuously monitors the movement of each peer's advertised window and the exact timings of data and acknowledgment packet arrivals. A number of rules (which are described in detail in a forthcoming paper) are used to determine if the peer, or the network, or even the filer is the bottleneck factor in data transfer. For instance, if the filer is sending data through a Gigabit interface but the receiver client does not advertise a window that is large enough for the estimated delay-bandwidth product of the connection, the client is flagged as "needing reconfiguration". If the receiver did initially (at the beginning of the connection) advertise a window that was sufficiently large, but subsequently this window shrank, this indicates that the client is unable to keep up with protocol processing at the maximum rate supported by the network, and this situation is flagged.

Cross-layer analysis is used to make the TCP logic aware as to what time-periods of a TCP connection are "interesting" from the point of view of performance auto-diagnosis of the type described in the previous paragraph. For example, the beginning of a large NFS read may indicate the beginning of an "interesting" time period for an otherwise idle TCP connection. Protocol augmentation (using ICMP ping based average RTT measurement) is used to estimate the delay-bandwidth product of the path to various clients.

## 4.2 Extensibility

Data ONTAP contains an implementation of the Java Virtual Machine. Our approach to addressing the issue of extensibility is to write most of the auto-diagnosis system in Java. This provides us complete flexibility to change the auto-diagnosis logic in a new version, and support older versions of ONTAP. In a Java scenario, the auto-diagnosis logic ships as a collection of Java classes that reside on the root file system of the filer.

Note that the current version of the auto-diagnosis system is in C; we plan to use Java in the next major version of ONTAP. As mentioned earlier, a file containing constant and threshold values provides limited extensibility in the current implementation.

## 4.3 Implementation in other operating systems

Based on our knowledge of the internals of BSD-like general-purpose operating systems and our experience with the implementation of the NetApp Auto-diagnosis

System, it appears that it should be relatively straightforward to implement an auto-diagnosis subsystem based on the techniques presented in this paper in BSD-like systems. Like in ONTAP, a kernel thread can be used to implement continuous monitoring. The rules and thresholds used by the continuous monitoring logic can be chosen based on field information about the problem situations being targeted. The continuous monitoring logic can be partitioned into sub-logic blocks for each major OS subsystem.

The specific active tests to be implemented will also depend on the field problems being targeted. Protocol augmentation and cross-layer analysis can be used as guiding principles in the design of such tests. The implementation of active tests triggered automatically by the in-kernel auto-diagnosis logic should be relatively straightforward. If a command-based approach (like our diagnostic commands based approach) is to be used for user interaction, the BSD *kvm* kernel memory interface may be used for transferring information between the in-kernel continuous monitoring logic and user-land diagnostic commands. Alternatively, an approach based on the */proc* file system may be used.

Special interfaces will be needed for commands to trigger in-kernel active tests. Again, several alternative approaches are possible. The tests may be enumerated and made available as a series of system calls or different *iocls* for a special diagnostics pseudo-device. For some active tests, it might also be possible to write the tests entirely in user-space using low level kernel interfaces (e.g. raw IP sockets). Extensibility can be provided by implementing the kernel portion of the auto-diagnosis logic as one or more loadable kernel modules.

## 5 Performance and experience

In this section, we will briefly discuss the performance of the NetApp Auto-diagnosis System, and our experience with its effectiveness in making the task of debugging field problems simple.

The continuous monitoring subsystem of ONTAP takes very few resources. Its CPU overhead is less than 0.25% CPU, even on the slowest systems that we ship. The memory footprint is less than 400KB for a typical system. The time that the *netdiag* command takes depends on the configuration of the system and the load on the system. On our slowest filer that is configured with the maximum number of allowable network interfaces and is saturated with client load, *netdiag* takes no more than 15 seconds to execute. On most systems, it takes less than 5 seconds.

Specifically, on a F760 class filer (600 Mhz 21164 Alpha, 2GB RAM) configured with 4 network interfaces and under full client load, the CPU usage of auto-diagnosis continuous monitoring code is less than 0.1% CPU. On this system, *netdiag* takes approximately 4 sec-

onds to execute.

The version of ONTAP that contains the NetApp Auto-diagnosis System has only recently been made available to customers. However, since this version of ONTAP has not yet shipped to our customers in volume, we have not been able to see how well the auto-diagnosis subsystem is able to deal with real-life problems in the field. Instead, we have been forced to rely on a study in the laboratory. In this study we simulated a sample of field problem cases from our customer support call record database and measured the effectiveness of the auto-diagnosis system in solving the problems. For each case, we re-created the specific problem situation in the laboratory and measured the effectiveness of the auto-diagnosis logic.

We first looked at a sample of 961 calls that came in during the month of September 1999. This set did not include calls corresponding to hardware or software faults. We also did not consider calls that were related to general information about the product asked for by the customer. All other types of calls were considered. The month of September 1999 was the first month whose call data we did *not* include in our analysis of historical call record data while designing ONTAP's auto-diagnosis logic.

Of these 961 calls, 84 had something to do with the networking code and its interactions with the rest of ONTAP. Auto-diagnosis, when simulated on these cases, was able to auto-detect the problem cause for all but 12 of these calls, at a success-rate of 84.5%. The average time that it took the *netdiag* command to diagnose the problem was approximately 2.5 seconds. We did not even attempt to quantify the secondary effect on the customer's level of satisfaction that auto-diagnosis would cause due to the dramatic reduction in average problem diagnosis time.

Of the 12 calls on which auto-diagnosis did not diagnose, 7 were related to transient problems with external networking hardware, 1 was due to a NIC that was exhibiting very occasional errors and had needed re-seating and 4 were problems for which we did not have appropriate auto-diagnosis logic.

Of the 877 calls not corresponding to networking, we performed a static manual analysis in order to figure out which of these problems could be auto-diagnosed by the complete ONTAP auto-diagnosis system. This analysis was performed against a design description of the auto-diagnosis logic for other subsystems of ONTAP. Our study indicates that about three quarters of these problems could indeed be addressed by auto-diagnosis. Another 124 (about 20%) of these calls corresponded to problems whose diagnosis could be sped-up significantly by the partial auto-diagnosis information that the diagnosis system provided.

We repeated this simulation and analysis for calls that came in during October 1999. We considered 1023



cases, 97 networking and 926 other. Simulation of the networking cases indicated that auto-diagnosis could solve 88% of these. All but 5 of the networking problems that could not be auto-diagnosed were related to misconfigured clients. The rest were problems for which we have not yet developed appropriate auto-diagnosis logic. Static manual analysis of the non-networking cases indicated a success-rate of about 70%.

We also considered 500 randomly chosen samples from the customer call data from the months of November 1999 through February 2000. We repeated the above described analysis and simulation for these 500 calls. Our results for this sample of calls were very similar to the results for September and October 1999.

In summary, our historical call data seems to indicate that our auto-diagnosis system will be hugely successful in making a lot of problems that currently require human intervention to be automatically addressed. This should lead to a big reduction in the cost of handling customer calls because of a significant reduction in the number of calls per installed system. We were unable to directly quantify the increase in simplicity of the problem diagnosis process; the only (relatively weak) metric that we could quantify was turnaround time for the problem, with and without auto-diagnosis. This metric was at least three orders of magnitude lower for auto-diagnosis.

## 6 Related work

To place our work in context, we briefly survey other approaches to field problem diagnosis of computer systems, and how our work relates to these techniques.

### 6.1 Ad-hoc monitoring of UNIX and UNIX-like systems

As briefly described before, most UNIX and UNIX-like operating systems maintain a large number of statistics corresponding to various events that have occurred in the operation of the system. Access to these statistics and other configuration information is provided by a number of command interfaces. Problem diagnosis usually consists of manually obtaining appropriate statistics and perusing them for aberrant values.

System administrators in some organizations that use a large number of UNIX systems often use a set of home-grown (or commercially available) frameworks of automated scripts to obtain information from a large number of systems and analyse these values. There is a wealth of literature describing these tools [29, 10, 9, 2]. In some ways, this is similar to our technique of continuous monitoring. The information gathered by these automated scripts, however, is at the granularity at which the various operating systems export system information. This granularity is usually too coarse for extensive auto-diagnosis of the kind that we can perform inside the operating system kernel with reasonable system overhead. These en-

vironments are also limited in the types of active tests that they can perform for pin-pointing problems.

### 6.2 SNMP

The Simple Network Management Protocol (SNMP) [3] allows for the management of systems in a TCP/IP network within a coherent framework. In the SNMP world, network management consists of *network management stations*, called managers, communicating with the various systems in the network (hosts, routers, terminal servers etc.), called *network elements*. SNMP based management consists of three parts: 1) a Management Information Base (MIB) [18] that defines the various variables (both standardized and vendor-specific) that network elements maintain that can be queried and set by the manager, 2) a set of common structures and an identification schema, called the Structure of Management Information (SMI) [28], that is used to reference the variables in the MIB, and 3) the protocol with which managers and elements communicate, i.e., SNMP.

The system works as follows: The network managers periodically send queries to the elements to get the state of the various elements. Elements send *traps* to managers when certain events happen. The manager may analyse the information available to it via results of queries to build a picture of the health of the network and present this information to the human network manager in a variety of ways. Plugins that extend a managers functionality in a vendor-specific manner are available to handle vendor specific MIBs. An example of a commonly used manager is HP's OpenView [11].

The problem of using SNMP is in some ways similar to the problem of defining appropriate checks for our continuous monitoring system. The various system variables that are checked by continuous monitoring equations correspond to MIB variables. The auto-diagnosis checking logic corresponds to logic in the network manager plugin handling the vendor-specific MIBs. Thus issues that arise in defining the checks that a continuous monitoring system should execute also apply to the design of SNMP logic.

SNMP is different from our system in two main ways: First, SNMP does not really have a parallel for our active tests. A manager can manipulate a network element in some limited fashion, e.g., by using setting appropriate MIB variables. However, this is not nearly as general or as powerful as what can be done by an active test executing in the concerned system itself.

Second, the fact that SNMP depends on the network connectivity to be present between the network elements and the manager limits the types of problems that can be effectively auto-diagnosed by using SNMP. In particular, problems effecting network connectivity may not be easily diagnosed by SNMP.

In some ways the use of SNMP complements our ap-



proach. A system of auto-diagnosis using the techniques that we described earlier may be responsible for the "local" health of a system and its interactions with other networking entities that it communicates with. An SNMP based network management infrastructure may provide overall information about the health of a network using information gained by communication with network elements and their auto-diagnosis subsystems.

### 6.3 Problem diagnosis systems from the mainframe and telecommunications world

A number of papers and patents in the literature describe various components of semi-automatic problem diagnosis systems that were developed and used in the context of mainframe computer systems [14, 31, 22], other highly reliable systems [1, 17, 32] and the phone system [35, 21]. These systems used the technique of continuous monitoring of the health of the system. Events affecting the health of the system were fed into a decision tree based expert diagnosis system. The expert system used the input events to walk down its decision tree to narrow down the set of possible problematic situations that might be present.

The hardest part of building such a system was defining the set of events to be monitored and building the knowledge base (the decision tree) of the expert system. There is some literature that describes at an abstract level how such knowledge base rule-sets can be created for a specific system based on probabilistic data about events and problems [35, 34]. Presumably, in practice, these knowledge bases were created based on experience information gathered from the field.

In some ways the work that we describe in this paper is similar to this older work. We also use continuous monitoring and have a rule-set and use thresholds to trigger off further diagnosis steps including various kinds of active tests.

Our work differs from this older work in that it provides novel guiding principles and a certain structure to the problem of designing rule-sets, thresholds, causality in related diagnosis procedures and active tests. The four auto-diagnosis techniques that we present were designed based on our knowledge of common field problems and how the occurrence of such field problems effects the dynamics of modern layered operating systems. The technique of protocol augmentation directly targets problems that arise out of inadequate, incompletely specified or poorly implemented open network protocols. Such problems are much more widespread and important in today's network-centric open computing infrastructures than in older environments where communication was based on closed proprietary protocols.

## 7 Summary and future work

To summarize, we described some general techniques to enable *appliance-like* debugging of field problems of network appliances. These techniques formalize various ad-hoc debugging techniques that are used in manual debugging of system problems by human experts. These techniques also help in making the task of debugging hard problems manually much simpler and quicker than it currently is.

We have implemented these ideas in the Data ON-TAP operating system. Our laboratory studies primed with real historical case data seem to indicate that auto-diagnosis as a methodology is very viable and has the potential of greatly reducing the complexity of problem analysis that is exposed to the customer.

In terms of future work, we would like to expand our continuous monitoring logic to encompass more complicated problems. As mentioned earlier, we are in the process of making the auto-diagnosis system extensible and easy to re-configure; this problem has a number of interesting issues. It would also be interesting to see a new user-interface paradigm linked with the ideas discussed in this paper that can vary the amount of detail and complexity in the output of the system based on the expertise of the user.

While our discussion has focused on Data ON-TAP, from our experience it seems that most of the ideas described in this paper are directly applicable general-purpose operating systems. ON-TAP's network code is based on BSD, and much of our auto-diagnosis logic can be directly applied to any BSD based TCP/IP subsystem. We look forward to an application of some of these ideas to general-purpose operating systems.

## Acknowledgements

Lots of people at NetApp helped in the work described in this paper. Barbara Naden, Henk Bots, Devi Nagraj, Mark Smith, Diptish Datta, Brian Pawlowski, Janet Takami, Susan Whitford, Paul Norman and a large number of folks in NetApp's customer satisfaction department contributed in terms of useful ideas, discussion and feedback. We are also grateful to our shepherd Aaron Brown, and the anonymous USENIX reviewers for their valuable feedback.

## References

- [1] Alex Winokur and Joseph Shiloach and Amnon Ribak and Yuangeng Huang. Problem determination method for local area network system. US Patent 5539877, 1996.
- [2] M. Burgess and R. Ralston. Distributed resource administration using cfengine. *Software—Practice and Experience*, 27, 1997.
- [3] J. D. Case, M. S. Fedor, M. L. Schoffstall, and

- C. Davin. Simple Network Management Protocol (SNMP). RFC 1157, May 1990.
- [4] Cisco Local Director. <http://www.cisco.com/warp/public/cc/cisco/mkt/scale/locald/>.
  - [5] Cobalt CacheRaQ 2. <http://www.cobalt.com/products/cache/index.html>.
  - [6] Cobalt NASRaQ. <http://www.cobalt.com/products/nas/index.html>.
  - [7] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.
  - [8] e/BSD: BSDI Embedded Systems Technology. <http://www.BSDI.COM/products/eBSD/>.
  - [9] M. Gomberg, C. Stacey, and J. Sayre. Scalable, Remote Administration of Windows NT. In *Proceedings of the Second Large Installation Systems Administration of Windows NT Conference (LISA-NT)*, Seattle, WA, July 1999.
  - [10] S. Hansen and T. Atkins. Centralized System Monitoring With Swatch. In *Proceedings of the Seventh Systems Administration Conference (LISA)*, Monterey, CA, Nov. 1993.
  - [11] HP OpenView. <http://www.openview.hp.com/>.
  - [12] IBM SecureWay Network Dispatcher. <http://www.ibm.com/software/network/dispatcher/>.
  - [13] Intel InBusiness eMail Station. [http://www.intel.com/network/smallbiz/inbusiness\\_email.htm](http://www.intel.com/network/smallbiz/inbusiness_email.htm).
  - [14] L. Koved and G. Waldbaum. Improving Availability of Software Subsystems Through On-Line Error Detection. *IBM Systems Journal*, 25(1):105–115, 1986.
  - [15] P. J. Leach and D. C. Naik. A Common Internet File System (CIFS/1.0) Protocol. Internet Draft, Network Working Group, Dec. 1997.
  - [16] J. Liedtke, V. Panteleenko, T. Jaeger, and N. Islam. High-performance caching with the Lava hit-server. In *Proceedings of the USENIX 1998 Annual Technical Conference*, New Orleans, LA, June 1998.
  - [17] Luan J. Denny. Threshold alarms for processing errors in a multiplex communications system. US Patent 4817092, 1989.
  - [18] K. McCloghrie and M. T. Rose. Management Information Base for Network management of TCP/IP-based Internets: MIB-II. RFC 1213, Mar. 1991.
  - [19] Mirapoint Internet Message Server. <http://www.mirapoint.com/products/servers/index.asp>.
  - [20] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *Proc. of the 1996 USENIX Technical Conference*, pages 99–111, 1996.
  - [21] Mohammad T. Fatehi and Fred L. Heismann. Performance monitoring and fault location for optical equipment, systems and networks. US Patent 5296956, 1994.
  - [22] R. E. Moore. Utilizing the SNA Alert in the Management of Multivendor Networks. *IBM Systems Journal*, 27(1):15–31, 1988.
  - [23] D. Mosberger and L. L. Peterson. Making paths explicit in the scout operating system. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.
  - [24] Network Appliance – Products – Filers. <http://www.netapp.com/products/filer/>.
  - [25] Network Appliance – Products – NetCache. <http://www.netapp.com/products/netcache/>.
  - [26] Network Appliance Technical Library. <http://www.netapp.com/tech.library/>.
  - [27] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3: Design and Implementation. In *Proceedings of the USENIX 1994 Summer Technical Conference*, Boston, MA, June 1994.
  - [28] M. T. Rose and K. McCloghrie. Structure and Identification of management Information for TCP/IP-based Internets. RFC 1155, May 1990.
  - [29] E. Sorenson and S. R. Chalup. RedAlert: A Scalable System for Application Monitoring. In *Proceedings of the Thirteenth Systems Administration Conference (LISA)*, Seattle, WA, Nov. 1999.
  - [30] SPEC SFS97. <http://www.specbench.org/osg/sfs-97/>.
  - [31] T. P. Sullivan. Communications Network Management Enhancements for SNA Networks: An Overview. *IBM Systems Journal*, 22(1/2):129–142, 1983.
  - [32] Wing M. Chan. Data integrity checking with fault tolerance. US Patent 4827478, 1989.
  - [33] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated Volume 2*. Addison-Wesley, Reading, MA, 1995.
  - [34] Yuval Lirov and On C. Yue. Technique for producing an expert system for system fault diagnosis. US Patent 5107497, 1992.
  - [35] Yuval Lirov and Swaminathan Ravikumar and On-Ching Yue. Arrangement for automated troubleshooting using selective advice and a learning knowledge base. US Patent 5107499, 1992.

# Dynamic Function Placement for Data-intensive Cluster Computing

Khalil Amiri, David Petrou, Gregory R. Ganger, Garth A. Gibson  
Carnegie Mellon University  
{amiri+,dpetrou,ganger,garth}@cs.cmu.edu  
<http://www.pdl.cs.cmu.edu/>

## Abstract

*Optimally partitioning application and filesystem functionality within a cluster of clients and servers is a difficult problem due to dynamic variations in application behavior, resource availability, and workload mixes. This paper presents ABACUS, a run-time system that monitors and dynamically changes function placement for applications that manipulate large data sets. Several examples of data-intensive workloads are used to show the importance of proper function placement and its dependence on dynamic run-time characteristics, with performance differences frequently reaching 2–10X. We evaluate how well the ABACUS prototype adapts to run-time system behavior, including both long-term variation (e.g., filter selectivity) and short-term variation (e.g., multi-phase applications and inter-application resource contention). Our experiments with ABACUS indicate that it is possible to adapt in all of these situations and that the adaptation converges most quickly in those cases where the performance impact is most significant.*

## 1 Introduction

Effectively utilizing cluster resources remains a difficult problem for distributed applications. Because of the relatively high cost of remote versus local communication, the performance of a large number of these applications is sensitive to the distribution of their functions across the network. As a result, the effective use of cluster

resources requires not only load balancing, but also proper partitioning of functionality among producers and consumers. While software engineering techniques (e.g., modularity and object orientation) have given us the ability to partition applications into a set of interacting functions, we do not yet have solid techniques for determining where in the cluster each of these functions should run, and deployed systems continue to rely on complex manual decisions made by programmers and system administrators.

Optimal placement of functions in a cluster is difficult because the right answer is usually “it depends.” Specifically, optimal function placement depends on a variety of cluster characteristics (e.g., communication bandwidth between nodes, relative processor speeds among nodes) and workload characteristics (e.g., bytes moved among functions, instructions executed by each function). Some are basic hardware characteristics that only change when something fails or is upgraded, and thus are relatively constant for a given system. Other characteristics cannot be determined until application invocation time, because they depend on input parameters. Worst of all, many change at run-time due to an application changing phases or competition between concurrent applications over shared resources. Hence, any “one system fits all” solution will cause suboptimal, and in some cases disastrous, performance.

In this paper, we focus on an important class of applications for which clusters are very appealing: data-intensive applications that selectively filter, mine, sort, or otherwise manipulate large data sets. Such applications benefit from the ability to spread their data-

---

This research is supported by DARPA/ITO, through DARPA Order D306, and issued by Indian Head Division NSWC, under contract N00174-96-0002. Additional support was provided by the members of the PDC, including EMC, Hewlett-Packard, Hitachi, Intel, LSI Logic, Novell, Quantum, Seagate Technology, and 3COM.

parallel computations across the source/sink servers, exploiting the servers' computational resources and reducing the required network bandwidth. Effective function partitioning for these data-intensive applications will become even more important as processing power becomes ubiquitous, reaching devices and network-attached appliances. This abundance of processing cycles has recently led researchers to augment storage servers with support for executing application-specific code. We refer to all such servers, which may be Jini-enhanced storage appliances [25], much-evolved commodity active disks [1, 19, 24] or file servers allowing remote execution of applications, as *programmable storage servers*, or simply *storage servers*.

In addition to their importance, we observe that these data-intensive applications have characteristics that simplify the tasks involved with dynamic function placement. Specifically, these applications all move and process significant amounts of data, enabling a monitoring system to *quickly* learn about the most important inter-object communication patterns and per-object resource requirements. This information allows the run-time system to rapidly identify functions that should be moved to reduce communication overheads or resource contention. In our prototype system, called ABACUS, functions associated with particular data streams are moved back and forth between clients and servers in response to dynamic conditions. In our implementation, programmers explicitly partition the functions associated with data streams into distinct components, conforming to an intuitive object-based programming model. The ABACUS run-time system monitors the resource consumption and communication of these components, without knowing anything about their internals (black box monitoring). The measurements are used with a cost-benefit model to decide when to relocate components to more optimal locations.

In this paper, we describe the design and implementation of ABACUS and a set of experiments evaluating its ability to adapt to changing conditions. Specifically, we explore how well ABACUS adapts to variations in network topology, application cache access pattern, application data reduction (filter selectivity), contention over shared data, phases

in application behavior, and dynamic competition for resources by concurrent applications. Our preliminary results are quite promising: ABACUS often improves application response time by 2–10X. In all of our experiments, ABACUS selects the best placement for each function, “correcting” placement when the function is initially started on the “wrong” node. Further, ABACUS often outperforms any static one-time placement in situations where dynamic changes cause the proper placement to vary during an application's execution. ABACUS is able to effectively adapt function placement based on only black box monitoring, removing from programmers the burden of considering function placement.

The remainder of this paper is organized as follows. Section 2 discusses how ABACUS relates to prior work. Section 3 describes the design of ABACUS. Section 4 discusses the ABACUS programming model and several example applications built upon it. Section 5 describes the run-time system. Section 6 presents a variety of experiments to demonstrate the value of dynamic function placement and ABACUS's ability to effectively adapt to dynamic conditions. Section 7 summarizes the paper's contributions.

## 2 Related work

There exists a large base of excellent research and practical experiences related to code mobility and cluster computing—far too large to fully enumerate here. This section discusses the most relevant previous work on adaptive function placement and how it relates to ABACUS.

Several previous systems such as Coign and others [16, 22] have demonstrated that function placement decisions can be automated given accurate profiles of inter-object communication and per-object resource consumption. All of these systems use long-term histories to make good installation-time or invocation-time function placement decisions. ABACUS complements these previous systems by looking at how to dynamically adapt placement decisions to run-time conditions.

River [4] is a system that dynamically adjusts per-consumer rates to match production rates, and per-producer rates to meet consumption rate variations. Such adjustments allow it



to adapt to run-time non-uniformities among cluster systems performing the *same* task. ABACUS complements River by adapting function placement dynamically in the presence of multiple *different* tasks.

Equanimity is a system that, like ABACUS, dynamically balances service between a single client and its server [14]. ABACUS builds on this work by developing mechanisms for dynamic function placement in realistic cluster environments, which include such complexities as resource contention, resource heterogeneity, and workload variation.

Hybrid shipping [10] is a technique proposed to dynamically distribute query processing load between clients and servers of a database management system. This technique uses *a priori* knowledge of the algorithms implemented by the query operators to estimate the best partitioning of work between clients and servers. Instead, ABACUS applies to a wider class of applications by relying only on black-box monitoring to make placement decisions, without knowledge of the semantics or algorithms implemented by the application components.

Process migration systems such as Condor [7] and Sprite [8] developed mechanisms for coarse-grain load-balancing among cluster systems, but did not explicitly support fine-grain function placement or adapt to inter-function communication. Mobile programming systems such as Emerald [18] and Rover [17] do support fine-grain mobility of application objects, but they leave migration decisions to the application programmer. Similarly, mobile agent systems, such as Mole [26] and Agent Tcl [12], enable agent migration but do not provide algorithms or mechanisms to decide where agents should be placed. ABACUS builds on such work by providing run-time mechanisms that automate migration decisions.

### 3 Overview of ABACUS

To explore the benefits of dynamic function placement, we designed and implemented the ABACUS prototype system and ported several test applications to it. ABACUS consists of a programming model and a run-time system. Our goal was to make the programming model easy for application programmers to use. Further, we

wanted it to simplify the task of the run-time system in migrating functions and in monitoring the resources they consume. As for the run-time system, our goals were to improve overall performance, through effective placement, and to achieve low monitoring overhead. Moreover, it was designed to scale to large cluster sizes. Our first ABACUS prototype largely meets these goals.

The ABACUS programming model encourages the programmer to compose data-intensive applications from explicitly-migratable, functionally independent components or objects. These *mobile* objects provide explicit methods that *checkpoint* and *restore* their state during migration. At run-time, an application and filesystem can be represented as a graph of communicating mobile objects. This graph can be thought of as rooted at the storage servers by anchored (non-migratable) *storage objects* and at the client by an anchored *console* object. The storage objects provide persistent storage, while the console object contains the part of the application that must remain at the node where the application is started. Usually, the console part is not data intensive. Instead, it serves to interact with the user or the rest of the system at the start node and typically consists of the main function in a C/C++ program. This console part initiates invocations that are propagated by the ABACUS run-time to the rest of the graph.

As shown in Figure 1, the ABACUS run-time system consists of (i) a migration and location-transparent invocation component, or *binding manager* for short; and (ii) a resource monitoring and management component, or *resource manager* for short. The first component is responsible for the creation of location-transparent references to mobile objects, for the redirection of method invocations in the face of object migrations, and for enacting object migrations. Also, each machine's binding manager notifies the local resource manager of each procedure call to and return from a mobile object.

The resource manager uses the notifications to collect statistics about bytes moved between objects and about the resources used by the objects (*e.g.*, amount of memory allocated, number of instructions executed per byte processed). A resource manager also monitors the load on its local processor and the

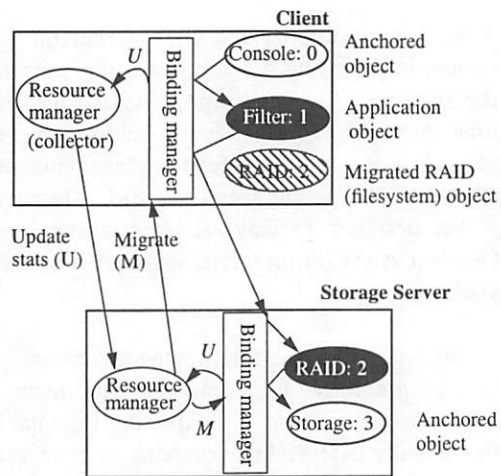


Figure 1: An illustration of an ABACUS object graph, the principal ABACUS components, and their interactions. This example shows a filter application accessing a striped file. Functionality is partitioned into objects. Dark ovals depict mobile objects, while clear ovals mark anchored objects. Inter-object method invocations are transparently redirected by the location transparent invocation component of the ABACUS run-time. This component also updates a local resource monitoring component on each procedure call and return from a mobile object (machine-local arrows labeled “U”). Clients periodically send digests of this collected information to the server. Resource managers at the server collect the relevant statistics and initiate migration decisions (arrows labeled “M”).

experienced stall time on network transfers to and from the storage servers that are actively accessed by local mobile objects. Server-side resource managers collect statistics from client-side resource managers and employ an analytic model to predict the performance benefit of moving to an alternative placement. The model also takes into account the cost of migrations, including the time wasted waiting until the object is quiescent and the time wasted for checkpointing the object, transferring its state, and restoring it on the target node. Using this analytic model, the server-side resource manager arrives at the placement with the best *net benefit*. If this placement is different from the current configuration, the necessary object migrations take place.

The ABACUS prototype is written in C++. We leverage the language’s object-oriented features to simplify writing our mobile objects. For our inter-node communication transport, we use DCE

RPC. Because the focus of our prototype is on function placement decisions, we do not address several important but orthogonal mobile code issues. For example, we sidestep the issues of code mobility [27] and dynamic linking [6] by requiring that all migratable modules be statically linked into an ABACUS process on both clients and servers. Further, issues of inter-module protection [28, 23, 9] are not addressed. While these will be important issues for production systems, they are tangential to the questions addressed in this paper.

## 4 Programming model

The ABACUS programming model has two principal aspects: *mobile objects*, which represent the unit of migration and placement, and an *iterative processing model*, which defines how mobile objects are composed into entire data processing applications.

### 4.1 Mobile objects

A mobile object in ABACUS is explicitly declared by the programmer as such. It consists of state and the methods that manipulate that state. A mobile object is required to implement a few methods to enable the run-time system to create instances of it and migrate it. Mobile objects are usually of medium granularity—they are not meant to be simple primitive types—performing a self-contained processing step that is data intensive, such as parity computation, caching, searching, or aggregation.

Mobile objects have private state that is not accessible to outside objects, except through the exported interface. The implementation of a mobile object is internal to that object and is opaque to other mobile objects and to the ABACUS run-time system. The private state consists of embedded objects and references to external objects. A mobile object is responsible for saving its private state, including the state of all embedded objects, when its `Checkpoint()` method is called by ABACUS. It is also responsible for reinstating this state, including the creation and initialization of all embedded objects, when the run-time system invokes the `Restore()` method, after it has been migrated to a new node. The `Checkpoint()` method saves the state to either an in-memory buffer or to a file. The `Restore()` method can reinstate the state

from either location. Both methods are invoked when there is no external invocation active within the mobile object.

Each storage server (i.e., a server with a data store) provides local storage objects exporting a flat file interface. Storage objects are accessible only at the server that hosts them and therefore never migrate. The migratable portion of the application lies between the storage objects on one side and the console object on the other. Applications can declare other objects to be non-migratable. For instance, an object that implements write-ahead logging can be declared by the filesystem as non-migratable, effectively anchoring it to the storage server where it is started (usually the server hosting the log).

## 4.2 Iterative processing model

Synchronous invocations start at the top-level console object and propagate down the object graph. Each invocation returns back to the console object with a result after a specified number of application records have been processed. Once an invocation returns to the console, objects in the graph are usually no longer active. Objects are activated again by the next *iteration*, which starts with a new invocation initiated by the console. Sometimes, objects may become “spontaneously” active, initiating invocations before any request is received from top-level objects. This occurs when objects perform background work (such as write-behind in a cache object), although that is not assumed to be the common mode of operation.

The amount of data moved in each invocation is an application-specific number of records, and not the entire file or data set at once. This iterative property is required by our monitoring and migration system. ABACUS accumulates statistics on return from method invocations for use in making object migration decisions. If the program makes a single procedure call down a stack of objects, ABACUS will not collect this valuable information until the end of the program, at which point any migration would be useless.

## 4.3 Examples

To stress the ABACUS programming model and evaluate the benefit of adaptive function

placement, we have implemented an object-based distributed filesystem and a few data intensive applications. We describe them in this section and report on their performance in Section 6.

### Object-based distributed filesystem.

Applications often require a variety of services from the underlying storage system. ABACUS enables filesystems to be composed of explicitly migratable objects, each providing storage services such as reliability (*e.g.*, RAID), caching, and application-specific functionality. This approach was pioneered by the stackable and composable filesystem work [13, 21] and by the Spring object-oriented operating system [20].

The ABACUS filesystem provides coherent file and directory abstractions atop the flat file space exported by base storage objects. A file is associated with a stack of objects when it is created representing the services that are bound to that file. For instance, only “important” files include a RAID object in their stack. When a file is opened, the top-most object is instantiated, which in turn instantiates all the lower level objects in the object graph. Access to a file always starts at the top-most object in the stack and the run-time system propagates accesses down to lower layers as needed.

The prototype filesystem is distributed. Therefore, it must contain, in addition to the layers that are typically found in local filesystems (such as caching and RAID), services to support inter-client file and directory sharing. In particular, the filesystem allows both file data and directory data (data blocks) to be cached and manipulated at trusted clients. Because multiple clients can be concurrently sharing files, we implement AFS style callbacks for cache coherence [15]. Similarly, because multiple clients can be concurrently updating directory blocks, the filesystem includes a timestamp-ordering protocol to ensure that updates performed at the clients are consistent before they are committed at the server. This scheme is highly scalable in the absence of contention because it does not require a lock server or any lock traffic. In Section 6.5, we describe how ABACUS automatically changes the concurrency control protocol during high contention to a locking scheme by simply adapting object placement.

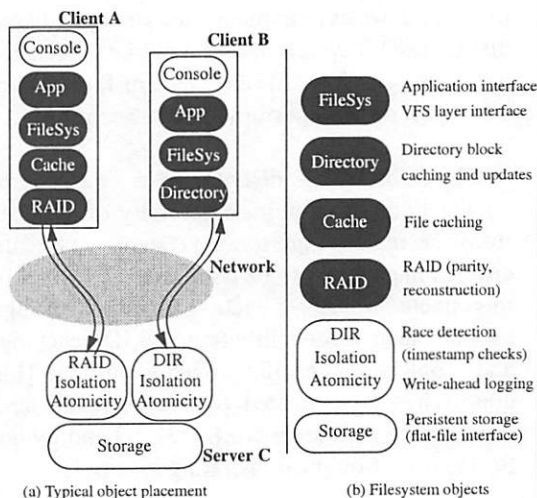


Figure 2: The architecture of the object-based distributed filesystem built atop ABACUS. The figure shows a typical file and directory object stack (a). The object placement shown is the default for high-bandwidth networks and trusted clients. Also shown are the component filesystem objects which are implemented to date and a brief description of their function (b).

By default, each file's graph consists of a filesystem object providing the VFS interface to applications, a cache object, an optional RAID 5 object, and one or more storage objects. To ensure that parity is not corrupted by races involving concurrent writes to the same stripe, a RAID isolation/atomicity object is anchored to each storage server. This object intercepts all reads and writes to the base storage object and verifies the consistency of updates before committing them. The protocols used by this isolation object are highly scalable and are described elsewhere [2]. The cache object keeps an index of a particular object's blocks in the shared cache kept by the ABACUS filesystem process. The RAID 5 object stripes and maintains parity for individual files across sets of storage servers. The storage objects provide flat storage and can be configured to use either the standard Linux `ext2` filesystem or CMU's Network-Attached Secure Disks (NASD) prototype [11] as backing store. Figure 2 shows a sketch of typical file and directory stacks.

Each directory's graph consists of a directory object, an isolation/atomicity object and a storage object. The directory object provides POSIX-like directory calls and caches directory

entries. The isolation/atomicity object provides support for both cache coherence and optimistic concurrency control, and also ensures the atomicity of multi-block writes to a directory. For performance reasons, the implementation of the isolation/atomicity object is specialized to directory semantics and is therefore different from the RAID 5 isolation/atomicity object described above. This object ensures cache coherence by interposing on read and write calls, installing callbacks on cached blocks during read calls and breaking relevant callbacks during writes. It ensures proper concurrency control among simultaneous updates by timestamping cache blocks [5] and exporting a special `CommitAction()` method that checks specified *readSets* and *writeSets* for conflicts.<sup>1</sup> Finally, the atomicity of multi-block writes is provided by ensuring that a set of blocks (possibly from differing objects and devices) are either updated in their entirety or not updated at all, by using a write-ahead log that is shared among all of the instances of the isolation/atomicity object.

The ABACUS filesystem can be accessed in two ways. First, applications that include ABACUS objects can directly append per-file object subgraphs onto their application object graphs for each file opened. Second, the ABACUS filesystem can be mounted as a standard filesystem, via VFS-layer redirection. Unmodified applications using the standard POSIX system calls can thus interact with the ABACUS filesystem. Although it does not allow legacy applications to be migrated, this second mechanism does allow legacy applications to benefit from the filesystem objects adaptively migrating beneath them.

**Object-based applications.** Data-intensive applications can be similarly decomposed into objects that perform operations such as search, aggregation, or data mining. Porting a data-intensive application, such as search, to ABACUS is straightforward. Most search applications already iterate over input data by invoking successive read calls to the filesystem and operating on a buffer at a time. Porting this

<sup>1</sup>The *readSet* (*writeSet*) consists of the list of blocks read (written) by the client. A directory operation such as `MkDir()` requires reading all the directory blocks to ensure the name does not exist then updating one block to insert the new name and inode number. The *readSet* in this case would contain all the directory blocks and their timestamps and the *writeSet* would contain the block that was updated.



kind of application simply requires encapsulating the filtering component of the search into a C++ object and writing checkpoint/restore methods for it. These methods are also relatively straightforward, since the state often consists of just the positions in the input and output files, and the contents of the current buffer.

## 5 Run-time system

The ABACUS run-time system consists of per-node binding managers and resource managers. Each binding manager is responsible for the instantiation of mobile objects and the invocation of their methods in a location-transparent manner (Section 5.1) and for the migration of objects between cluster nodes (Section 5.2). The resource managers collect statistics about resource usage and availability (Section 5.3) and use these measurements to adapt placement decisions to improve total application response time (Section 5.4).

### 5.1 Object instantiation and invocation

The two kinds of nodes in an ABACUS cluster are clients and servers. Servers are nodes on which at least one base storage object resides and clients are nodes that execute applications that access storage servers. Since servers can also execute applications, one storage server can potentially be a client of another server.

Applications instantiate mobile objects by making a request to the ABACUS run-time system. For example, when filtering a file, the application console object will request a filter object to be created. The run-time system creates the object in memory by invoking the new operator of the C++ run-time.

ABACUS also allocates and returns to the caller a network-wide unique run-time identifier, called a *rid*, for the new object.<sup>2</sup> The caller uses the *rid* to invoke the new mobile object. The *rid* acts as a layer of indirection, allowing objects to refer to other objects without knowing their current location. The ABACUS binding manager mediates method invocations and uses *rids* to forward them to the object's current

<sup>2</sup>The *rid* is a network-wide identifier, which is generated by concatenating the node identifier where the object is created and a local object identifier that is unique within that node.

location. ABACUS maintains the information necessary to perform inter-object invocations in a per-node hash table that maps an *rid* to a (*node*, *object\_reference\_within\_node*) pair. As mobile objects move between nodes, this table is updated to reflect the new node and the new object reference at that node. The *rid* is passed as the first argument of each method invocation, allowing the system to properly redirect method calls.

At run-time, this web of objects constitutes a graph whose nodes represent objects and whose edges represent invocations between objects. For objects in the same address space, invocations are implemented via procedure calls, and data is passed without any extra copies. For objects communicating across machines or address spaces, remote procedure calls (RPCs) are employed.

### 5.2 Object migration

In addition to properly routing object calls, ABACUS binding managers are responsible for enacting migration. Consider migrating a given object from a *source node* to a *target node*. First, the binding manager at the source node blocks new calls to the migrating object. Then, the binding manager waits until all active invocations in the migrating object have drained (returned). Migration is cancelled if this step takes too long.

Next, the object is checkpointed locally by invoking its `Checkpoint()` method. The object allocates an in-memory buffer to store its state or writes to the filesystem if the checkpoint size is large. This state is then transferred and restored on the storage node. Then, the location tables at the source and target nodes are updated to reflect the new location. Finally, invocations are unblocked and are redirected to the proper node via the updated location table. This procedure extends to migrating subgraphs of objects.

Location tables are not always accurate. Instead, they provide hints about an object's location, which may become stale. Nodes other than the source and target that have cached hints about an object's location will not be updated when a migration occurs. However, stale data is detected and corrected when these nodes attempt to invoke the object at the old node. At that time, the old node notifies them that the object

has migrated.

For scalability reasons, nodes are not required to maintain forwarding pointers for objects that they have hosted at some point in the past. Consequently, the old node may not be able to inform a caller of the current location of an object. In this case, the old node will redirect the caller to the node at which the object originated, called the *home node*. The home node can be easily determined because it is encoded in the object's rid. The home node always has up-to-date information about the object's location because during each migration its location table is updated in addition to the tables at the source and target nodes. Because objects usually move between a client (an object's home node) and one of the servers, extra messaging is not usually required to update location tables during migration.

### 5.3 Resource monitoring

The run-time system uses its intermediary role in redirecting calls to collect all the necessary statistics. By only interposing monitoring code at procedure call and return from mobile objects, ABACUS does not slow down the execution of methods within a mobile object. This section explains how the needed statistics are collected.

On a single node, threads can cross the boundaries of multiple mobile objects by making method invocations that propagate down the stack. The resource manager must charge the time a thread spends computing or blocked to the appropriate object. Similarly, it must charge any allocated memory to the proper object. The ABACUS run-time collects the required statistics over the previous  $H$  seconds of execution, which we refer to as the observation window. We describe how some of these statistics are collected:

**Data flow graph.** The bytes moved between objects are monitored by inspecting the arguments on procedure call and return from a mobile object. The number of bytes transferred between two objects is then recorded in a timed data flow graph. This graph maintains moving averages of the bytes moved between every pair of communicating objects in a graph. These data flow graphs are of tractable size because most data-intensive applications do the bulk of

their processing in a stream-like fashion through a small stack of objects.

**Memory consumption.** ABACUS monitors the amount of memory dynamically allocated by an object as follows. On each procedure call or return from a mobile object, the pid of the thread making the call is recorded. Thus, for at any point in time, the run-time system knows the currently processing mobile object for each active thread in that address space. Wrappers around each memory allocation routine (e.g., `malloc`, `free`) inspect the pid of the thread invoking the memory allocation routine and use that pid to determine the current object. This object is then charged for the memory that was allocated or freed.

**Instructions executed per byte.** Given the number of bytes processed by an object, computing the instructions/byte amounts to monitoring the number of instructions executed by the object during the observation window. Given the processing rate on a node, this amounts to measuring the time spent computing within an object. We use a combination of the Linux interval timers and the Pentium cycle counter to keep track of the time spent processing within a mobile object.

**Stall time.** To estimate the amount of time a thread spends stalled in an object, one needs more information than is currently provided by the POSIX system timers. We extend the `getitimer/setitimer` system calls to support a new type of timer, which we denote `ITIMER_BLOCKING`. This timer decrements whenever a thread is blocked and is implemented as follows: When the kernel updates the system, user, and real timers for the active thread, it also updates the blocking timers of any threads in the queue that are marked as blocked (`TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`).

### 5.4 Dynamic Placement

The resource manager on a given server seeks to perform the migrations that will result in the minimal average application response time across all the applications that are accessing it. This amounts to figuring out what subset of objects executing currently on clients can benefit most from computing closer to the data. Migrating an

object to the server could potentially reduce the amount of stall time on the network, but it could also extend the time the object spends computing if the server's processor is overloaded.

Resource managers at the servers use an analytic model to determine which objects should be migrated from the clients to the server and which objects, if any, should be migrated back from the server to the clients. The analytic model considers alternative placement configurations and selects the one with the best *net benefit*, which is the difference between the benefit of moving to that placement and the cost of migrating to it. This net benefit represents the estimated reduction in execution time over the next  $H$  seconds.

A migration is actually enacted only if the server-side resource manager finds a new placement whose associated net benefit exceeds a configurable threshold,  $B_{T_{thresh}}$ . This threshold value is used to avoid migrations that chase small improvements, and it can be set to reflect the confidence in the measurements and the models used by the run-time system. Server-side resource managers do not communicate with one another to figure out the globally optimal placement. A server-side resource manager decides on the best alternative placement considering only the application streams that access it. This design decision was taken for robustness and scalability reasons.

The details of the computation required to estimate the net benefit are discussed in an associated technical report [3]. Here, we outline the intuition behind the computation. The server-side resource manager receives the per-object measurements described above. It also receives statistics about the client processor speed and current load and collects similar measurements about the local system and locally executing objects. Given the data flow graph between objects, the measured stall time of client-side objects, and the latency of the client-server link, the model estimates the change in stall time if an object changes location. Given the instructions per byte and the relative load and speed of the client/server processors, it estimates the change in execution time if the object changes placement. In addition to the change in execution time for the migrated object, the model also estimates the change in execution time for the

other objects executing at the target node (as a result of the increased load on the node's processor). When considering different object placements, we treat the memory available at the server as a fixed constraint. Together, the changes in stall time and execution time amount to the benefit of the new placement. In computing this benefit, our analytic model assumes that history will repeat itself over the next window of observation (the next  $H$  seconds). The cost associated with a placement is estimated as the sum of a fixed cost (the time taken to wait until the object is quiescent) plus the time to transfer the object's state between source and destination nodes. This latter value is estimated from the size of the checkpoint buffer and the bandwidth between the nodes.

## 6 Performance evaluation

In this section, we show how performance depends on the appropriate placement of function. The subsections that follow give increasingly difficult cases where ABACUS can adapt function placement even when the correct location is hard or impossible to anticipate at design-time. This includes scenarios in which the objects' correct location is based on hardware characteristics, application run-time parameters, application data access patterns, and inter-application contention over shared data. This also includes scenarios that stress adaptation under dynamic conditions: phases of application behavior and contention by multiple applications. We could not perform a fair comparison of applications running on ABACUS to those running on a network filesystem, such as NFS, because our filesystem implementation differs from that of Linux's NFS. The differences give ABACUS applications advantages that have little to do with adaptive function placement.

### 6.1 Evaluation environment

Our evaluation environment consists of eight clients and four storage servers. All twelve nodes are standard PCs running RedHat Linux 5.2 and are equipped with 300MHz Pentium II processors and 128MB of main memory. None of our experiments exhibited significant paging activity. Each server contains a single Maxtor 84320D4 IDE disk drive (4GB, 10ms average seek, 5200RPM, up to 14MB/s media transfer rate). Our environment consists of two networks:

a switched 100Mbps Ethernet, which we refer to as the *SAN* (server-area network) and a shared 10Mbps segment, which we refer to as the *LAN* (local-area network). All four storage servers are directly connected to the SAN, whereas four of the eight clients are connected to the SAN (called SAN clients), and the other four clients reside on the LAN (the LAN clients). The LAN is bridged to the SAN via a 10Mbps link. While these networks are of low performance by today's standards, their relative speeds are similar to those seen in high-performance SAN and LAN environments (Gbps in the SAN and 100Mbps in the LAN).

The bar graphs in the following sections adhere to a common format. Each graph shows the elapsed time of several configurations of an experiment with a migrating object. For each configuration, we report three numbers: the object (1) statically located at the client, (2) beginning at the client, but with ABACUS dynamically monitoring the system and potentially migrating the object, and (3) statically at the storage server. Graphs with confidence intervals report averages over five runs with 90% confidence. We have intentionally chosen smaller benchmarks to underscore ABACUS's ability to adapt quickly. We note that the absolute benefit achieved by dynamic function placement is often a function of the duration of a particular benchmark, and that longer benchmarks operating on larger files would amortize adaptation delays more thoroughly. Throughout the experiments in this section, the observation window,  $H$ , was set to 1 second, and the threshold benefit,  $B_{Thresh}$ , was set to 30% of the observation window.

## 6.2 Adapting to network topology/speed

**Issue.** Network topology/speed dictate the relative importance of cross-network communication relative to server load. Here we evaluate the ability of ABACUS to adapt to different network topologies. We default to executing function at clients to offload contended servers. However, ABACUS moves function to a server if a client would benefit and the server has the requisite cycles. The goal is to see whether ABACUS can decide when the benefit of server-side execution due to the reduction in network stall time exceeds the possible slowdown due to slower server-side processing.

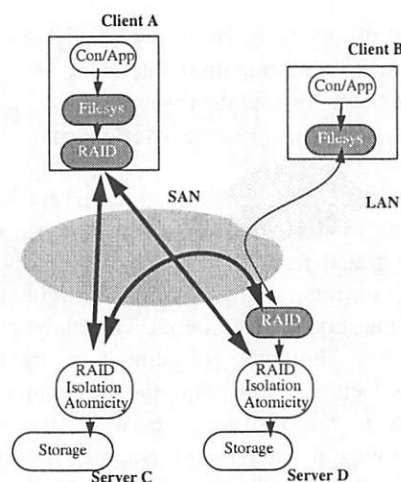


Figure 3: This figure shows a file bound to an application accessing a RAID object which maintains per-file parity code and accesses storage objects running on the storage devices. We show one binding of the stack (Client A) where the RAID object runs at the client, and another binding (Client B) where the RAID object runs at one of the storage devices. Thicker lines represent more data being moved. The appropriate configuration is dependent on the bandwidth available between the client and storage devices. If the client LAN is slow, Client B's partitioning would lead to lower access latencies.

**Experiment.** Software RAID is an example of a function that moves a significant amount of data and often touches every byte (computes the bitwise XOR of the contents of multiple blocks). Files in the ABACUS filesystem can be bound to a RAID object that provides storage striping and fault-tolerance. The RAID object maintains parity on a per-file basis, stripes data across multiple storage servers, and is distributed to allow concurrent accesses to shared stripes by clients by using a timestamp-based concurrency control protocol [2]. The RAID object can execute at either the client nodes or the storage servers. The object graph used by files for this experiment is shown in Figure 3.

The proper placement of the RAID object largely depends on the performance of the network connecting the client to the storage servers. Recall that a RAID small write involves four I/Os, two to pre-read the old data and parity and two to write the new data and parity. Similarly, when a disk failure occurs, a block read requires reading all the blocks in a stripe and XORing them together to reconstruct the failed



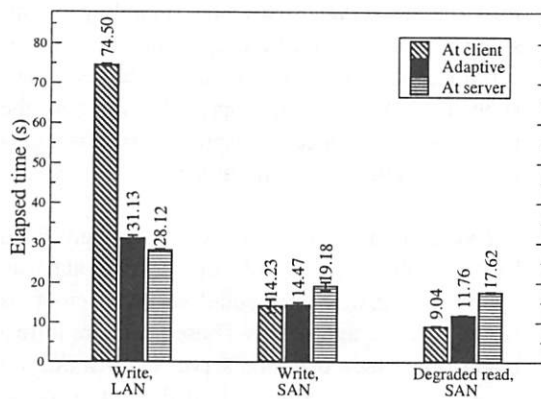


Figure 4: This figure shows the results of our RAID benchmark. Contention on the server's CPU resources make client-based RAID more appropriate, except in the LAN case, where the network is the bottleneck.

data. This can result in substantial network traffic between the RAID object and the storage servers.

We construct two workloads to evaluate RAID performance on ABACUS. The first consists of two clients writing two separate 4MB files sequentially. The stripe size is 5 (4 data + parity) and the stripe unit is 32 KB. The second workload consists of the two clients reading the files back in degraded mode (with one disk marked failed).

**Results.** As shown in Figure 4, executing the RAID object at the server improves RAID small write performance in the LAN case by a factor of 2.6 over executing the object at the host. The performance of the experiment when ABACUS adaptively places the object is within 10% of optimal. Conversely, in the SAN case, executing the RAID object locally at the client is 1.3X faster because the client has a lower load and is able to perform the RAID functionality more quickly. Here, ABACUS arrives within 1% of this value. The advantage of client-based RAID is slightly more pronounced in the more CPU-intensive degraded read case, in which the optimal location is almost twice as fast as at the server. Here, ABACUS arrives within 30% of optimal. In every instance, ABACUS automatically selects the best location for the RAID object.

### 6.3 Adapting to run-time parameters

**Issue.** Applications can exhibit drastically different behavior based on run-time parameters.

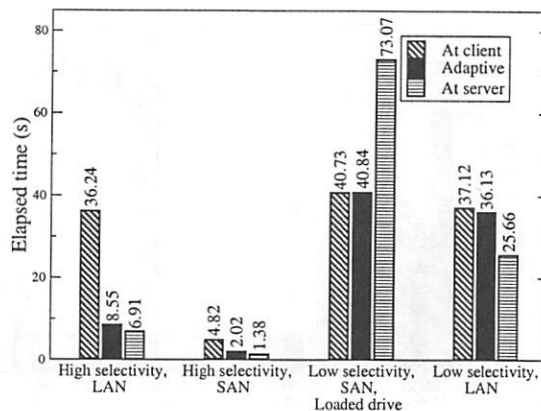


Figure 5: The performance of our filter benchmark is shown in this figure. Executing the filter at the storage server is advantageous in all but the third configuration, in which the filter is computationally expensive and runs faster on the client, has more CPU resources available.

In this section, we show that the data being accessed by a filter (which is set by an argument) determines the appropriate location for the filter to run. For example, there's a drastic difference between `grep kernel Bible.txt` and `grep kernel LinuxBible.txt`.

**Experiment.** As data sets in large-scale businesses continue to grow, an increasingly important user application is high-performance search, or data filtering. Filtering is often a highly selective operation, consuming a large amount of data and producing a smaller fraction. We constructed a synthetic filter object that returns a configurable percentage of the input data to the object above it. Highly selective filters represent ideal candidate for execution close to the data, so long as storage resources are available.

In this experiment, we varied both the filter's selectivity and CPU consumption from low to high. We define selectivity as  $(1 - \text{output}/\text{input})$ . A filter labeled low selectivity outputs 80% of the data that it reads, while a filter with high selectivity outputs only 20% of its input data. A filter with low CPU consumption does the minimal amount of work to achieve this function, while a filter with high CPU consumption simulates traversing large data structures (*e.g.*, the finite state machines of a text search program like `grep`).

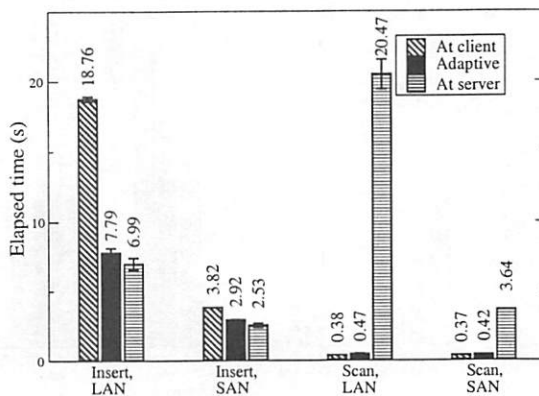


Figure 6: The figure shows that client-side caching is essential for workloads exhibiting reuse (Scan), but causes pathological performance when inserting small records (Insert). ABACUS automatically enables and disables the client caching by placing the cache object at the client or at the server.

**Results.** Figure 5 shows the elapsed time to read and filter a 16MB file in a number of configurations. In the first set of numbers, ABACUS migrates the filter from the client to the storage server, coming within 25% of the ideal case, which is over 5X better than filtering at the client. Similarly, ABACUS migrates the filter in the second set. While achieving better performance than statically locating the filter at the client, ABACUS reaches only within 50% of optimal because the time required for ABACUS to migrate the object is a bigger fraction of total run-time. In the third set, we run a computationally expensive filter. We simulate a loaded or slower storage server by making the filter twice as expensive to run on the storage server. Here, the filter executes 1.8X faster on the client. ABACUS correctly detects this case and keeps the filter on the client. Finally, in the fourth set of numbers, the value of moving is too low for ABACUS to deem it worthy of migration. Recall that the migration threshold is 30%, and note that this applies to the *estimated* benefit computed by ABACUS and not the real benefit.

#### 6.4 Adapting to data access patterns

**Issue.** Client-side caches in distributed file and database systems often yield dramatic reduction in storage access latencies because they avoid slow client networks, increase the total amount of memory available for caching, and reduce the

load on the server. However, enabling client-side caching can yield the opposite effect under certain access patterns. In this subsection, we show that ABACUS appropriately migrates the per-file cache object in response to data access patterns via black-box monitoring.

**Experiment.** Caching in ABACUS is provided by a mobile cache object. Consider an application that inserts small records into files stored on a storage server. These inserts require a read of the block from the server (an *installation read*) and then a write-back of the entire block. Even when the original block is cached, writing a small record in a block requires transferring the entire contents of each block to the server. Now, consider an application reading cached data. Here, we desire the cache to reside on the client.

We carried out the following experiments to evaluate the impact of and ABACUS's response to application access patterns. In the first benchmark, *table insert*, the application inserts 1,500 128byte records into a 192KB file. An insert writes a record to a random location in the file. In the second benchmark, *table scan*, the application reads the 1,500 records back, again in random order. The cache, which uses a block size of 8KB, is large enough for the working set of the application. Before recording numbers, the experiment was run to warm the cache.

**Results.** As shown in Figure 6, locating the cache at the server for the insert benchmark is 2.7X faster than at a client on the LAN, and 1.5X faster than at a client on the SAN. ABACUS comes within 10% of optimal for the LAN case, and within 15% for the SAN case. The difference is due to the relative length of the experiments, causing the cache to migrate relatively late in the SAN case (which runs for only a few multiples of the observation window). The table scan benchmark highlights the benefit of client-side caching when the application workload exhibits reuse. In this case, ABACUS leaves the cache at the client, cutting execution time over caching at the server by over 40X and 8X for the LAN and SAN tests respectively.

#### 6.5 Adapting to contention over shared data

**Issue.** Filesystem functionality, such as

caching or namespace updates/lookups, is often distributed to improve scalability [15]. When contention for the shared objects between clients is low, executing objects at the client(s) accessing them yields higher scalability and better cache locality. When contention over a shared object increases, a server-based execution becomes more efficient. In this case, client invocations are serialized locally on the server, avoiding the overhead of retries over the network. This kind of adaptation also solves performance cliffs caused by false sharing in distributed file caches. When several clients are writing to ranges in a file that happen to share common blocks, the invalidation traffic can degrade performance so that write-through to the server would be preferable.

**Experiment.** We chose a workload that performs directory inserts in a shared namespace as our contention benchmark. Directories in ABACUS present a hierarchical namespace like all UNIX filesystems and are implemented using the object graph shown in Figure 7.

When clients access disjoint parts of the directory namespace (i.e.: there are no concurrent conflicting accesses), the optimistic scheme in which concurrency control checks are performed by the isolation object (recall Section 4.3) works well. Each directory object at a client maintains a cache of the directories accessed frequently by that client, making directory reads fast. Moreover, directory updates are minimally cheap because no metadata pre-reads are required, and no lock messaging is performed. Further, offloading the bulk of the work from the server results in better scalability and frees storage devices to execute demanding workloads from competing clients. When contention is high, however, the number of retries and cache invalidations seen by the directory object increases, potentially causing several round-trip latencies per operation. When contention increases, we desire the directory object to migrate to the storage device. This would serialize client updates through one object, thereby eliminating retries.

We constructed two benchmarks to evaluate how ABACUS responds to different levels of directory contention. The first is a high contention workload, where four clients insert 200 files each in a shared directory. The second is a low contention workload where four

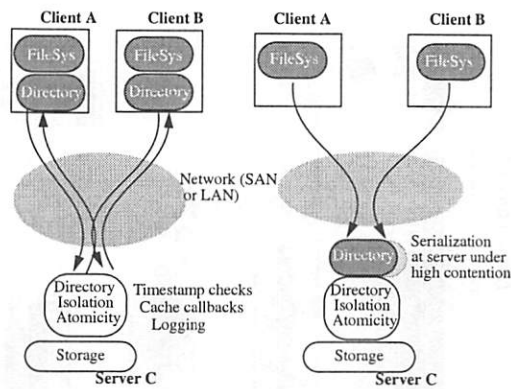


Figure 7: This figure shows directory updates from multiple contending clients. While distributing directory management to clients is beneficial under low contention, under high contention it results in a flurry of retries per directory operation. When the object is moved to the storage device, multiple client requests are serviced by (multiple threads in) the same object, serializing them locally without the cost of multiple cross-network retries.

clients insert 200 files each in private (unique) directories.

**Results.** As shown in Figure 8, ABACUS reduces execution time for the high contention workload by migrating the directory object to the server. In the LAN case, ABACUS is within 10% of the optimal. The optimal is 8X better than locating the directory object at the host. ABACUS comes within 25% of optimal for the high contention, SAN case (which is 2.5X better than the worst case). ABACUS estimates that moving it closer to the isolation object would make retries cheaper. It adapts more quickly in the LAN case because the estimated benefit is greater. ABACUS had to observe far more retries and revalidation traffic on the SAN case before deciding to migrate the object.

Under low contention, ABACUS makes different decisions in the LAN and SAN cases, migrating the directory object to the server in the former and not migrating it in the latter. We started the benchmark from a cold cache, causing many installation reads. Hence, in the case where there is little contention and the application is running over the LAN, ABACUS estimates that migrating the directory object to the storage server is worthwhile, because it avoids the latency of the low-speed LAN. However, in the SAN case, the network is

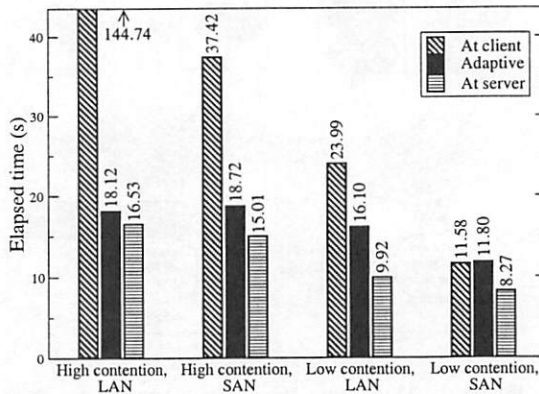


Figure 8: This figure shows the time to execute our directory insert benchmark under different levels of directory contention. ABACUS migrates the directory object in all but the fourth case.

	Insert	Scan	Insert	Scan	Tot.
Client	26.0	0.4	28.3	0.4	55.2
Adapt.	11.7	7.2	12.1	3.5	34.5
Server	7.8	29.2	7.7	26.0	70.7
Opt.	7.8	0.4	7.7	0.4	16.3

Table 1: Inter-application phases. This table shows the performance of a multiphasic application in the static cases and under ABACUS. The application goes through an insert phase, followed by a scan phase, back to inserting, and concluding with another scan. The table shows the completion time in seconds of each phase under each scheme.

fast enough that ABACUS cost-benefit model estimates the installation read network cost to be limited. Indeed, the results show that the static client and storage server configurations for the SAN case differ by less than 30%, our migration threshold. This benchmark does not exhibit a case where client-side placement is better because our client population is of limited size (only four nodes). Note also that the directory objects from different clients need not all migrate to the server at the same time. The server can decide to migrate them independently, based on its estimates of the migration benefit for each client. Correctness is ensured even if only *some* objects migrate, because all operations are verified to have occurred in timestamp order by the underlying isolation/atomicity object.

## 6.6 Adapting to application phases

**Issue.** Having established that optimal placement depends on several system and workload characteristics, we further note that these characteristics change with time on most systems. In this subsection, we are concerned with characteristics that vary with algorithmic changes within the lifetime of the application. Applications rarely exhibit the same behavior or consume resources at the same rate throughout their lifetimes. Instead, an application may change phases at a number of points during its execution in response to input from a user or a file or as a result of algorithmic properties. Multiphasic applications make a particularly compelling case for the dynamic function relocation that ABACUS provides.

**Experiment.** To explore multiphasic behavior, we revisit our file caching example. Specifically, we run a benchmark that does an insert phase, followed by scanning, followed by inserting, and concluding with another scan phase. The goal is to determine whether the benefit estimates at the server will eject an application that changed its behavior after being moved to the server. Further, we wish to see whether ABACUS recovers from bad history quickly enough to achieve adaptation that is useful to an application that exhibits multiple contrasting phases.

**Results.** Table 1 shows that ABACUS migrates the cache to the appropriate location based on the behavior of the application over time. First, ABACUS migrates the cache to the server for the insert phase. Then, ABACUS ejects the cache object from the server when it detects that the cache is being reused by the client. Both static choices lead to bad performance in alternating phases. Consequently, ABACUS outperforms both static cases—by 1.6X compared to the client case, and by 2X compared to the server case. The optimal row refers to the minimum execution time picked alternatively from the client and server cases. We see that ABACUS is approximately twice as slow as the optimal. This is to be expected, as this extreme scenario changes phases fairly rapidly.

## 6.7 Adapting to competition

**Issue.** Shared storage server resources are rarely dedicated to serving one workload. An



additional complexity addressed by ABACUS is provisioning storage server resources to competing clients. Toward reducing global application execution time, ABACUS resolves competition among objects that would execute more quickly at the server by favoring those objects that would derive a greater benefit from doing so.

**Experiment.** In this experiment, we run two filter objects on a 32MB file on our LAN. The filters have different selectivities, and hence derive different benefits from executing at the storage server. In detail, Filter 1 produces 60% of the data that it consumes, while Filter 2, being the more selective filter, outputs only 30% of the data it consumes. The storage server's memory resources are restricted so that it can only support one filter at a time.

**Results.** Figure 9 shows the cumulative progress of the filters over their execution, and the migration decisions made by ABACUS. The less selective Filter 1 is started first. ABACUS shortly migrates it to the storage server. Soon after, we start the more selective Filter 2. Shortly thereafter, ABACUS migrates the highly selective Filter 2 to the server, kicking back the other to its original node. The slopes of the curves show that the filter currently on the server runs faster than when not, but that Filter 2 derives more benefit since it is more selective. Filters are migrated to the server after a noticeable delay because the estimated benefit was close to the configured threshold. Longer history windows will amortize the migration cost over a longer window of benefit, resulting in migration occurring sooner. In general, the history window should be at least long enough to capture many iterations up and down the object stack, so that the statistics collected by ABACUS are representative of application behavior.

ABACUS does place a run-time overhead compared to traditional implementations. A filter implemented and running on ABACUS runs up to 25% slower than one implemented directly atop of the Unix Filesystem, in the case where no migrations occur.<sup>3</sup> Furthermore, ABACUS can, under pathological conditions, result in worse performance than either static

<sup>3</sup>The size of the file filtered was 8 MB. We believe that part of this overhead can be eliminated with a more optimized implementation.

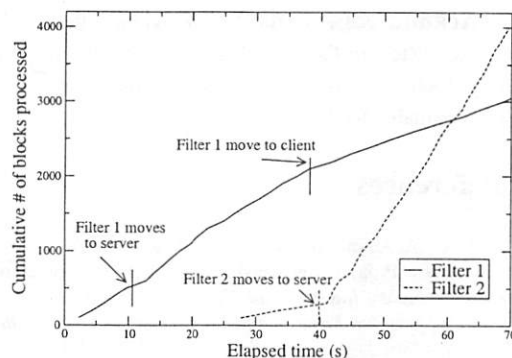


Figure 9: This figure plots the cumulative number of blocks searched by two filters versus elapsed time. ABACUS's competition resolving algorithm successfully chooses the more selective Filter 2 over the Filter 1 for execution at the storage server.

placement. Consider the case of an application that completely changes behavior right after ABACUS decides to migrate it, inducing a migration back to the original node, only to change its behavior again and start another cycle. In this case, the application will always be placed on the "wrong" node and will incur additional migration costs that are linear with the migration frequency, which is about once every history window. This worst case behavior is currently bounded by noticing objects that rapidly ping-pong back and forth between locations and anchoring them in one default placement until the application terminates.

## 7 Conclusions

In this paper, we demonstrate that optimal function placement depends on system and workload characteristics that are impossible to predict at application design or installation time. We propose a dynamic approach where function placement is continuously adapted by a run-time system based on resource usage and availability. Measurements demonstrate that placement can be decided based on black-box monitoring of application objects, in which the system is oblivious to the function being implemented. Preliminary evaluation shows that ABACUS, our prototype system, can improve application response time by 2–10X. These encouraging

results indicate a promising future for this approach.

**Acknowledgements.** We would like to thank John Wilkes, Richard Golding, David Nagle, our shepherd Christopher Small, and our anonymous reviewers for their valuable feedback.

## References

- [1] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 81–91, San Jose, CA, Oct. 1998.
- [2] K. Amiri, G. Gibson, and R. Golding. Highly concurrent shared storage. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, Taipei, Taiwan, Republic of China, Apr. 2000.
- [3] K. Amiri, D. Petrou, G. Ganger, and G. Gibson. Dynamic function placement in active storage clusters. Technical Report CMU-CS-99-140, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1999.
- [4] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 10–22, Atlanta, GA, May 1999.
- [5] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proceedings of the 6th Conference on Very Large Databases*, Oct. 1980.
- [6] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, Dec. 1995.
- [7] A. Bricker, M. Litzkow, and M. Livny. Condor Technical Summary. Technical Report 1069, University of Wisconsin—Madison, Computer Science Department, Oct. 1991.
- [8] F. Douglass and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software—Practice & Experience*, 21(8):757–785, Aug. 1991.
- [9] B. Ford, M. Hibler, J. Lepreau, P. Tullman, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, October 28–31, 1996. Seattle, WA, pages 137–151, Oct. 1996.
- [10] M. J. Franklin, B. T. Jónsson, and D. Kossmann. Performance tradeoffs for client-server query processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 25, 2 of *ACM SIGMOD Record*, pages 149–160, New York, June 4–6 1996.
- [11] G. A. Gibson, D. F. Nagle, W. Courtright, N. Lanza, P. Mazaitis, M. Unangst, and J. Zelenka. NASD scalable storage systems. In *Proceedings of the USENIX '99 Extreme Linux Workshop*, June 1999.
- [12] R. S. Gray. Agent Tcl: A flexible and secure mobile agent system. In *Proceedings of the Fourth Annual Tcl/Tk Workshop*, pages 9–23, Monterey, Cal., July 1996.
- [13] J. S. Heidemann and G. J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, Feb. 1994.
- [14] E. H. Herrin, II and R. A. Finkel. Service rebalancing. Technical Report CS-235-93, Department of Computer Science, University of Kentucky, May 18 1993.
- [15] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), Feb. 1988.
- [16] G. C. Hunt and M. L. Scott. The Coign automatic distributed partitioning system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, Feb. 1999.
- [17] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek. Mobile computing with the rover toolkit. *IEEE Transactions on Computers: Special issue on Mobile Computing*, Mar. 1997. <http://www.pdos.lcs.mit.edu/rover/>.
- [18] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, Jan. 1988.
- [19] K. Keeton, D. Patterson, and J. Hellerstein. A case for intelligent disks (IDISKS). *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(3), 1998.
- [20] Y. Khalidi and M. Nelson. Extensible File Systems in Spring. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 1–14, Dec. 1993.
- [21] O. Krieger and M. Stumm. HFS: A performance-oriented flexible file system based on building-block compositions. *ACM Transactions on Computer Systems*, 15(3):286–321, Aug. 1997.
- [22] J. Michel and A. van Dam. Experience with distributed processing on a host/satellite graphics system. In *Computer Graphics (SIGGRAPH '76 Proceedings)*, pages 190–195, July 1976.
- [23] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, Seattle, Wa., Oct. 1996.
- [24] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of the 24th VLDB Conference*. Very Large Data Base Endowment, August 1998.
- [25] Seagate. Jini: A Pathway for Intelligent Network Storage, 1999. Press release, <http://www.seagate.com/corp/vpr/literature/papers/jini.shtml>.
- [26] M. Straer, J. Baumann, and F. Hohl. Mole – a Java based mobile agent system. In *2nd ECOOP Workshop on Mobile Object Systems*, pages 28–35, Linz, Austria, July 1996.
- [27] T. Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, Sept. 1997.
- [28] R. Wahbe, S. Lucco, and T. Anderson. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, Dec. 1993.

# Scalable Content-aware Request Distribution in Cluster-based Network Servers

Mohit Aron

Darren Sanders

Peter Druschel

Willy Zwaenepoel

*Department of Computer Science, Rice University, Houston, TX 77005*

## Abstract

We present a scalable architecture for content-aware request distribution in Web server clusters. In this architecture, a level-4 switch acts as the point of contact for the server on the Internet and distributes the incoming requests to a number of back-end nodes. The switch does *not* perform any content-based distribution. This function is performed by *each* of the back-end nodes, which may forward the incoming request to another back-end based on the requested content.

In terms of scalability, this architecture compares favorably to existing approaches where a front-end node performs content-based distribution. In our architecture, the expensive operations of TCP connection establishment and handoff are distributed among the back-ends, rather than being centralized in the front-end node. Only a minimal additional latency penalty is paid for much improved scalability.

We have implemented this new architecture, and we demonstrate its superior scalability by comparing it to a system that performs content-aware distribution in the front-end, both under synthetic and trace-driven workloads.

## 1 Introduction

Servers based on clusters of workstations or PCs are the most popular hardware platform used to meet the ever growing traffic demands placed on the World Wide Web. This hardware platform combines a cutting edge price—performance ratio in the individual server nodes with the promise of perfect scalability as additional server nodes are added to

meet increasing demands. However, in order to actually attain scalable performance, it is essential that scalable mechanisms and policies for request distribution and load balancing be employed. In this paper, we present a novel, scalable mechanism for content-aware request distribution in cluster based Web servers.

State-of-the-art cluster-based servers employ a specialized front-end node, which acts as a single point of contact for clients and distributes requests to back-end nodes in the cluster. Typically, the front-end distributes requests such that the load among the back-end nodes remains balanced. With *content-aware request distribution*, the front-end additionally takes into account the content or type of service requested when deciding which back-end node should handle a given request.

Content-aware request distribution can improve scalability and flexibility by enabling the use of a partitioned server database and specialized server nodes. Moreover, previous work [6, 26, 31] has shown that by distributing requests based on cache affinity, content-aware request distribution can afford significant performance improvements compared to strategies that only take load into account.

While the use of a centralized request distribution strategy on the front-end affords simplicity, it can limit the scalability of the cluster. Very fast layer-4 switches are available that can act as front-ends for clusters where the request distribution strategies do not consider the requested content [11, 21]. The hardware based switch fabric of these layer-4 switches can scale to very large clusters. Unfortunately, layer-4 switches cannot efficiently support content-aware request distribution because the latter requires layer-7 (HTTP) processing on the front-end to determine the requested content. Conventional, PC/workstation based front-ends, on the other hand, can only scale to a relatively small num-

ber of server nodes (less than ten on typical Web workloads [26].)

In this paper, we investigate scalable mechanisms for content-aware request distribution. We propose a cluster architecture that decouples the request distribution strategy from the *distributor*, which is the front-end component that interfaces with the client and that distributes requests to back-ends. This has several advantages: (1) it dramatically improves the scalability of the system by enabling multiple distributor components to co-exist in the cluster, (2) it improves cluster utilization by enabling the distributors to reside in the back-end nodes, and (3) it retains the simplicity afforded by a centralized request distribution strategy.

The rest of the paper is organized as follows. Section 2 presents some background information for the rest of the paper. In Section 3, we discuss the cluster configurations currently used for supporting content-aware request distribution and we provide experimental results that quantify limitations in their scalability. Section 4 describes our scalable cluster design for supporting content-aware request distribution. We discuss our prototype implementation in Section 5 and Section 6 presents experimental results obtained with the prototype. Related work is covered in Section 7 and Section 8 presents conclusions.

## 2 Background

This section provides background information on content-aware request distribution and the existing mechanisms that support it.

### 2.1 Content-aware Request Distribution

Content-aware request distribution is a technique employed in cluster-based network servers, where the request distribution strategy takes into account the service/content requested when deciding which back-end node should serve a given request. In contrast, the purely load-based schemes like *weighted round-robin* (WRR) used in most commercial high performance cluster servers [11, 21] distribute incoming requests in a round-robin fashion, weighted

by some measure of load on the different back-end nodes.

The potential advantages of content-aware request distribution are: (1) increased performance due to improved hit rates in the back-end's main memory caches, (2) increased secondary storage scalability due to the ability to partition the server's database over the different back-end nodes, and (3) the ability to employ back-end nodes that are specialized for certain types of requests (e.g., audio and video). Locality-aware request distribution (LARD) is a specific strategy for content-aware request distribution that focuses on the first of the advantages cited above, namely improved cache hit rates in the back-ends [6, 26]. LARD improves cluster performance by simultaneously achieving load balancing and high cache hit rates at the back-ends.

In order to inspect the content of the requests, a TCP connection must be established with the client *prior* to assigning the request to a back-end node. This is because, with content-aware request distribution, the nature and target<sup>1</sup> of the client's request influences the assignment. Therefore, a mechanism is required that allows a chosen back-end node to serve a request on a TCP connection that was established elsewhere in the cluster. For reasons of performance, security and interoperability, it is desirable that this mechanism be transparent to the client. We discuss such mechanisms in the next subsection.

### 2.2 Mechanisms

A simple client-transparent mechanism for supporting content-aware request distribution is a *relaying front-end* depicted in Figure 1. An HTTP proxy running on the front-end accepts client connections, and maintains persistent connections with all the back-end nodes. When a request arrives on a client connection, the connection is assigned according to a request distribution strategy (e.g., LARD), and the proxy forwards the client's request message on the appropriate back-end connection. When the response arrives from the back-end node, the front-end proxy forwards the data on the client connection, buffering the data if necessary. The principal advantage of this approach is its simplicity and the

<sup>1</sup>The term *target* refers to a Web document, specified by a URL and any applicable arguments to the HTTP GET command.



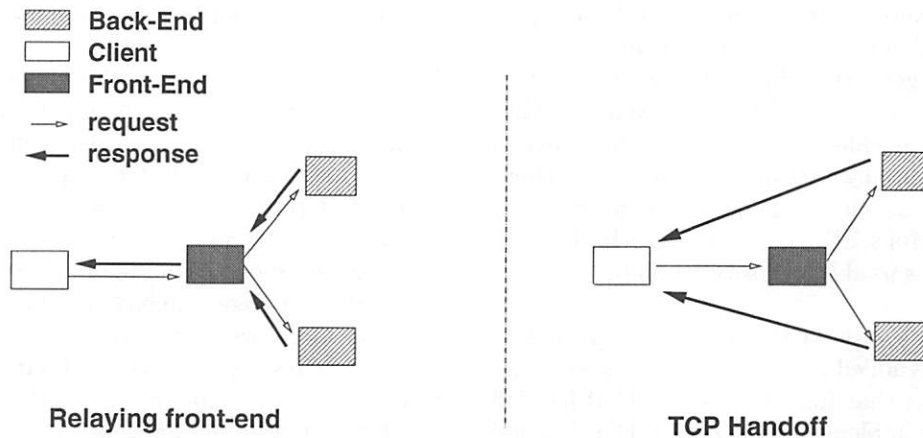


Figure 1: Mechanisms for request distribution

ability to be deployed without requiring any modification of the operating system kernel on any cluster node. The primary disadvantage, however, is the overhead incurred in forwarding all the response data from the back-ends to the clients, via the proxy application.

TCP splicing [15, 12] is an optimization of the front-end relaying approach where the data forwarding at the front-end is done directly in the operating system kernel. This eliminates the expensive copying and context switching operations that result from the use of a user-level application. While TCP splicing has lower overhead than front-end relaying, the approach still incurs high overhead because all HTTP response data must be forwarded via the front-end node. TCP splicing requires modifications to the OS kernel of the front-end node.

The TCP handoff [26] mechanism was introduced to enable the forwarding of back-end responses directly to the clients without passing through the front-end as an intermediary. This is achieved by handing off the TCP connection established with the client at the front-end to the back-end where the request was assigned. TCP handoff effectively serializes the state of the existing client TCP connection at the front-end, and instantiates a TCP connection with that given starting state at the chosen back-end node. The mechanism remains transparent to the clients in that the data sent by the back-ends appears to be coming from the front-end and any TCP acknowledgments sent by the client to the front-end are forwarded to the appropriate back-end. TCP handoff requires modifications to both the front-end and back-end nodes, typically via a loadable kernel module that plugs into a general-

purpose UNIX system.

While TCP handoff affords substantially higher scalability than TCP splicing by virtue of eliminating the forwarding overhead of response data, our experiments with a variety of trace-based workloads indicate that its scalability is in practice still limited to cluster sizes of up to ten nodes. The reason is that the front-end must establish a TCP connection with each client, parse the HTTP request header, make a strategy decision to assign the connection, and then serialize and handoff the connection to a back-end node. The overhead of these operations is substantial enough to cause a bottleneck with more than approximately ten back-end nodes on typical Web workloads.

### 3 Scalability of a single Front-end

Despite the use of splicing or handoff, a single front-end node limits the scalability of clusters that employ content-based request distribution. This section presents experimental results that quantify the scalability limits imposed by a conventional, single front-end node.

To measure the scalability of the splicing and TCP handoff mechanisms, we conducted experiments with the configurations depicted in Figure 1. Our testbed consists of a number of client machines, connected to a cluster server. The cluster nodes are 300MHz Intel Pentium II based PCs with 128MB of memory. All machines are configured with the FreeBSD-2.2.6 operating system.

The requests were generated by a HTTP client program designed for Web server benchmarking [8]. The program generates HTTP requests as fast as the Web server can handle them. Seven 166 MHz Pentium Pro machines configured with 64MB of memory were used as client machines. The client machines and all cluster nodes are connected via switched 100Mbps Ethernet. The Apache-1.3.3 [3] Web server was used at the server nodes.

For experiments with TCP handoff, a loadable kernel module was added to the OS of the front-end and back-end nodes that implements the TCP handoff protocol. The implementation of the TCP handoff protocol is described in our past work [6, 26]. For splicing, a loadable module was added to the front-end node. Persistent connections were established between the front-end node and the back-end web-servers for use by the splicing front-end.

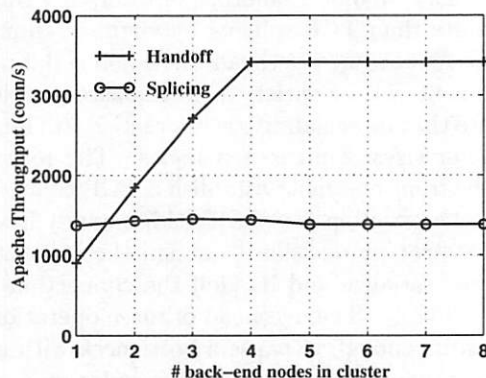


Figure 2: Throughput, 6 KB requests

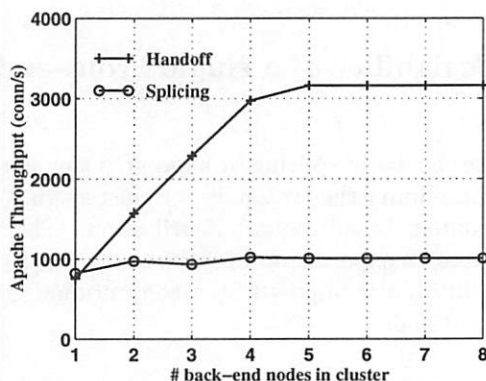


Figure 3: Throughput, 13 KB requests

In the first experiment, the clients repeatedly request the same Web page of a given size. Under this artificial workload, the LARD policy behaves like WRR: it distributes the incoming requests among

all back-end nodes in order to balance the load.

Figures 2 and 3 compare the cluster throughput with the handoff versus the splicing mechanism as a function of the number of back-end nodes in the cluster, and for requested Web page sizes of 6 KB and 13 KB, respectively. These two page sizes correspond to the extrema of the range of average HTTP transfer sizes reported in the literature [24, 4]. Since the requested page remains cached in the servers' main memory with this synthetic workload, the server displays very high throughput, thus fully exposing scalability limitations in the front-end request distribution mechanism.

The results show that the TCP handoff mechanism scales to four back-end nodes, while splicing is already operating at front-end capacity with only one server node. In either case, the scalability is limited because the front-end CPU reaches saturation.

For the 6 KB files, the performance of splicing exceeds that of handoff at a cluster size of one node. The reason is that splicing uses persistent connections to communicate with the back-end servers, thus shifting the per-request connection establishment overhead to the front-end, which results in a performance advantage in this case. For larger cluster sizes and large files size, this effect is more than compensated by the greater efficiency of handoff, and by the fact that splicing saturates the front-end.

Additionally, it should be noted that with the larger page size (13 KB), the throughput with splicing degrades more than that with handoff (27% versus 7%, respectively). This is intuitive because with splicing, the higher volume of response data has to pass through the front-end, while with handoff, the front-end only incurs the additional cost of forwarding more TCP acknowledgments from the client to the back-ends.

In general, the maximal throughput achieved by the cluster with a single front-end node is fixed for any given workload. For example, the throughput with the handoff mechanism is determined by (a) the rate at which connections can be established and handed off to back-end nodes and (b) the rate at which TCP acknowledgments can be forwarded to back-end nodes. With slow back-end webservers and/or with workloads that cause high per-request overhead (e.g., frequent disk accesses) at the back-end nodes, the maximum throughput afforded by a

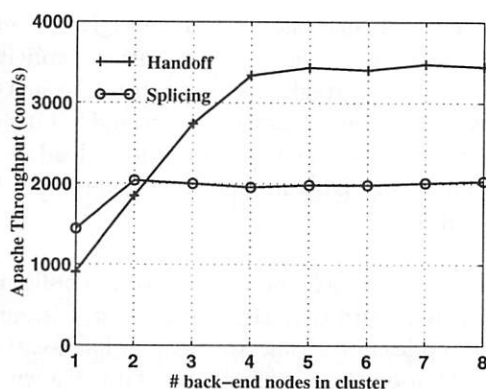


Figure 4: Throughput, IBM trace

back-end is lower and the front-end is able to support more back-end nodes before becoming a bottleneck. With efficient back-end web servers and/or workloads with low per-request overheads, a smaller number of back-ends can be supported.

The results shown above were obtained with a synthetic workload designed to expose the limits of scalability. Our next experiment uses a realistic, trace-based workload and explores the scalability of handoff and splicing under real workload conditions.

Figure 4 shows the achieved throughput using handoff versus splicing on a trace based workload derived from www.ibm.com's server logs (details about this trace are given in Section 6). As in the previous experiments, the handoff scales linearly to a cluster size of four nodes. The splicing mechanism scales to two back-end nodes on this workload. Thus, splicing scales better on this workload than on the previous, synthetic workload. The reason is that the IBM trace has an average page size of less than 3 KB<sup>2</sup>. Since splicing overhead is more sensitive to the average page size for the reasons cited above, it benefits from the low page size more than handoff.

The main result is that both splicing and handoff only scale to a small number of back-end nodes on realistic workloads. Despite the higher performance afforded by TCP handoff, its peak throughput is limited to about 3500 conn/s on the IBM trace and it does not scale well beyond four cluster nodes. In Section 6 we show that a software based layer-4 switch implemented using the same hardware can

<sup>2</sup>This is substantially less than the average Web page size reported by several studies. The likely reason is that the content designers of this high-volume site have minimized the size of the most popular pages for performance reasons.

afford a throughput of up to 20,000 conn/s. Therefore, the additional overhead imposed by content-aware request distribution reduces the scalability of the system by an order of magnitude.

The limited scalability of content-aware request distribution cannot be easily overcome through the use of multiple front-end nodes. Firstly, employing multiple front-end nodes introduces a secondary load balancing problem among the front-end nodes. Mechanisms like round-robin DNS are known to result in poor load balance. Secondly, many content-aware request distribution strategies (for instance, LARD) require centralized control and cannot easily be distributed over multiple front-end nodes.

In Section 4 we describe the design of a scalable content-aware request distribution mechanism that maintains centralized control. Results in Section 6 indicate that this cluster is capable of achieving an order of magnitude higher performance than existing approaches.

## 4 Scalable Cluster Design

This section addresses the scalability problem with content-aware request distribution. We identify the bottlenecks and propose a configuration that is significantly more scalable.

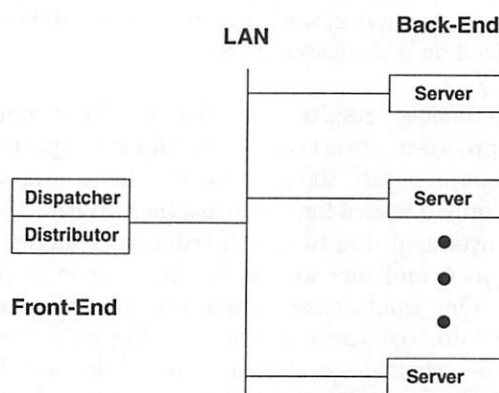


Figure 5: Cluster components

Figure 5 shows the main components comprising a cluster configuration with content-aware request distribution and a single front-end. The *dispatcher* is the component that implements the request distribution strategy; its task is to decide which server node should handle a given request. The *distrib-*

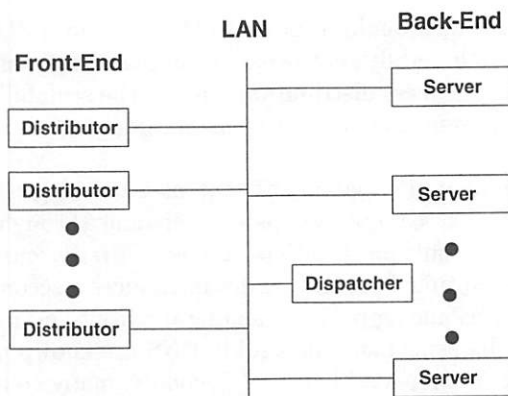


Figure 6: Multiple front-ends

utor component interfaces with the client and implements the mechanism that distributes the client requests to back-end nodes; it implements either a form of TCP handoff or the splicing mechanism. The *server* component represents the server running at the back-end and is responsible for processing HTTP client requests.

A key insight is that (1) the bulk of the overhead at the front-end node is incurred by the distributor component, not the dispatcher; and, (2) the distributor component can be readily distributed since its individual tasks are completely independent, while it is the dispatcher component that typically requires centralized control. It is intuitive then, that a more scalable request distribution can be achieved by distributing the distributor component over multiple cluster nodes, while leaving the dispatcher centralized on a dedicated node.

Experimental results show that for TCP handoff, the processing overhead for handling a typical connection is nearly 300  $\mu$ sec for the distributor while it is only 0.8  $\mu$ sec for the dispatcher. With splicing, the overhead due to the distributor is larger than 750  $\mu$ sec and increases with the average response size. One would expect then, that distributing the distributor component while leaving only the dispatcher centralized should increase the scalability of the request distribution by an order of magnitude. Our results presented in Section 6 confirm this reasoning.

Figure 6 shows a cluster configuration where the distributor component is distributed across several front-end nodes while the dispatcher resides on a dedicated node. In such a configuration with multiple front-end nodes, a choice must be made as

to which front-end should receive a given client request. This choice can be made either explicitly by the user with strategies like *mirroring*<sup>3</sup>, or in a client transparent manner using DNS round-robin. However, these approaches are known to lead to poor load balancing [20], in this case among the front-end nodes.

Another drawback of the cluster configuration shown in Figure 6 is that efficient partitioning of cluster nodes into either front-end or back-end nodes depends upon the workload and is not known *a priori*. For example, for workloads that generate significant load on the back-end nodes (e.g, queries for online databases), efficient cluster utilization can be achieved by using a few front-end nodes and a large number of back-end nodes. For other workloads, it might be necessary to have a larger number of front-end nodes. A suboptimal partitioning, relative to the prevailing workload, might result in low cluster utilization, i.e, either the front-end nodes become a bottleneck when the back-end nodes are idle or vice versa.

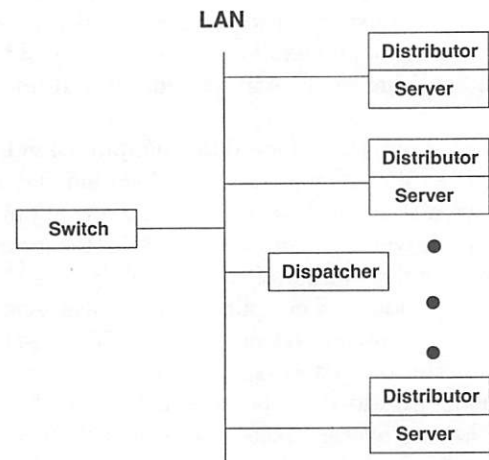


Figure 7: Co-located distributors and servers

Figure 7 shows an alternate cluster design where the distributor components are co-located with the server components. As each cluster node hosts both the distributor and the server components, the cluster can be efficiently utilized irrespective of the workload. As in the cluster of Figure 6, the replication of the distributor component on multiple cluster nodes eliminates the bottleneck imposed by a centralized distributor. In addition, a front-end consisting of a commodity layer-4 switch is employed

<sup>3</sup>With mirroring, the clients are aware of the multiple front-ends in the cluster and explicitly choose where to send their requests.



that distributes incoming client requests to one of the distributor components running on the back-end nodes, in such a way that the load among the distributors is balanced.

Notice that the switch employed in this configuration does not perform content-aware request distribution. It merely forwards incoming packets based on layer-4 information (packet type, port number, etc.). Therefore, a highly scalable, hardware based commercial Web switch product can be used for this purpose [21, 11]. In Section 6, we present experimental results with a software based layer-4 switch that we developed for our prototype cluster.

A potential remaining bottleneck with this design is the centralized dispatcher. However, experimental results presented in Section 6 show that a centralized dispatcher implementing the LARD policy can service up to 50,000 conn/s on a 300MHz Pentium II machine. This is an order of magnitude larger than the performance of clusters with a single front-end, as shown in Section 3.

In the next section, we provide a detailed description of our prototype implementation. In Section 6 we present experimental results that demonstrate the performance of our prototype.

## 5 Prototype Implementation

This section describes the implementation and operation of our prototype cluster. Section 5.1 gives an overview of the various components of the cluster. Section 5.2 provides details on the implementation of a software based layer-4 switch that we implemented for interfacing the cluster with the clients. Section 5.3 describes the sequence of operations in the cluster upon receiving a client request. Section 5.4 discusses techniques employed in our implementation to improve the scalability of the dispatcher by reducing the overhead of communicating with other cluster nodes.

### 5.1 Overview

The cluster consists of a number of nodes connected by a high speed LAN. As we mentioned in Section 4, there are three main components in the cluster – dis-

patcher, distributor and server. The interface design for these components is such that any cluster node can contain one or more of these components. This implies that any of the cluster configurations of Figure 5, Figure 6 or Figure 7 can be realized by placing the components appropriately on the cluster nodes.

The communication between components on different cluster nodes is realized using persistent TCP control connections that are created during the cluster initialization. A control connection, thus, exists between any two nodes of the cluster and multiplexes the messages exchanged between any two components. These connections also serve to detect node failures.

Owing to the superior performance afforded by the TCP handoff protocol as compared to splicing, our prototype distributor employs the handoff protocol. The server component consists of the user-level server application and an enhanced network protocol stack in the kernel capable of accepting connections using the handoff protocol. The server application can be any off-the-shelf web server (e.g., Apache [3], Zeus [30]) and requires no change for operation in the cluster. The distributor is also implemented as an enhanced protocol stack, and resides wholly in the kernel. Similarly, the dispatcher component also resides in the kernel for efficiency. The kernel changes required to implement the cluster components are added using a loadable kernel module for the FreeBSD-2.2.6 OS.

Our prototype implementation supports both HTTP/1.0 as well as HTTP/1.1 persistent connections (P-HTTP [24]). Support for P-HTTP is similar to that described in our earlier work [6]. As this paper focuses on scalability issues in the cluster, we only consider HTTP/1.0 connections in the experimental results presented in this paper.

A layer-4 switch is used that receives all requests from clients and forwards them to the distributors. With the switch, the distributed nature of the cluster becomes completely transparent to the clients. We next describe the implementation of this switch.

### 5.2 Layer-4 Switch

We implemented a fast software based layer-4 switch to be used as the front-end node. Even though hardware based, highly scalable layer-4 switches are com-

mercially available, we decided to implement a software based switch for two reasons: we did not have a commercial layer-4 switch available to us, and we wanted to explore what level of scalability could be achieved with a software based switch.

The switch maintains a small amount of state for each client connection being handled by the cluster. This state is used to determine the cluster node where packets from the client are to be forwarded. Additionally, the switch maintains the load on each cluster node based on the number of client connections handled by that node.

Upon seeing a connection establishment packet (TCP SYN packet), the switch chooses a distributor on the least loaded node in the cluster. Subsequent packets from the client are sent to the same node unless an *update* message instructs the switch to forward packets to some other node (update messages are sent by the distributor after a TCP connection handoff). Upon connection termination, a *close* message is sent to the switch by the cluster node that handles the connection and is used for discarding connection state at the switch.

All outgoing data from the cluster is sent directly to the clients and does not pass through the switch. Only the packets sent by the clients are received and forwarded by the switch. To improve switch performance, the forwarding module in the switch avoids interrupts and uses soft timer based polling to receive network packets [5, 25]. In Section 6, we report the forwarding throughput of this switch.

Using a front-end layer-4 switch (as opposed to using, for instance, DNS round-robin to distribute requests among the server nodes) offers several important advantages. The first is increased security. By hiding the back-end nodes of the cluster from the outside world, would-be attackers cannot directly attack these nodes. The second advantage is availability. By making the individual cluster nodes transparent to the client, failed or off-line server nodes can be replaced transparently. Finally, when combined with TCP handoff, the use of a switch increases efficiency, as ACK packets from clients need not be forwarded by the server node that originally received a request.

Of course, a front-end switch forms a single point of failure in the cluster. There are, however, a number of possible solutions to this problem, such as using a hot-swappable spare.

### 5.3 Cluster Operation

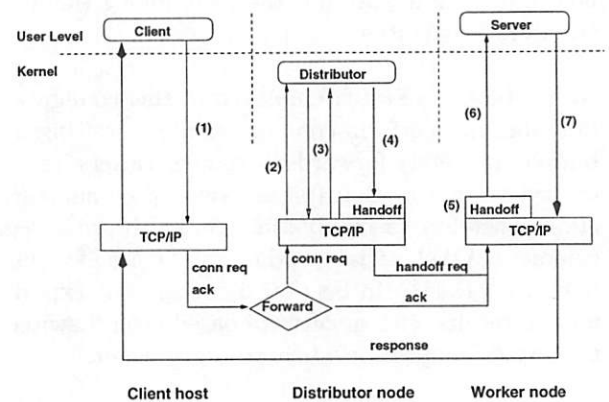


Figure 8: Operation with DNS round-robin

Figure 8 shows the operation of the cluster. For simplicity, we assume here that clients directly contact a distributor (for instance, via DNS round-robin). The figure shows the user-level processes and protocol stacks at the client and the distributor and server nodes. The client application (e.g., Web browser) is unchanged and runs on an unmodified standard operating system.

The figure illustrates the cluster operations from the time the client sends the request, to the instant when it receives a response from the cluster. (1) the client process (e.g., Netscape) uses the TCP/IP protocol to connect to the chosen distributor, (2) the distributor component accepts the connection and parses the client request, (3) the distributor contacts the dispatcher (not shown in the figure) for the assignment of the request to a server, (4) the distributor hands off the connection using the TCP handoff protocol to the server chosen by the dispatcher, (5) the server takes over the connection using its handoff protocol, (6) the server application at the server node accepts the created connection, and (7) the server sends the response directly to the client. Any TCP acknowledgments sent by the client to the distributor are forwarded to the server node using a *forwarding module* at the distributor node.

The operation in Figure 8 assumes that the server component chosen by the dispatcher resides on a different node than the distributor. If the server is on the same node as the distributor, then the handoff is accomplished efficiently using procedure calls in the kernel.

The operation of the cluster with a layer-4 switch is similar to that in Figure 8. However, there are some important differences. First, the choice of the distributor is made by the switch, using the WRR strategy. Second, after a connection is handed off by the distributor, the switch is notified and the subsequent forwarding of TCP acknowledgments to the corresponding server node is handled by the switch.

## 5.4 Batching Requests

As the dispatcher component is centralized, its performance determines the scalability of the cluster. We demonstrate in Section 6.1 that the bulk of the overhead imposed on the dispatcher node is associated with communication with other cluster nodes. The key to achieving greater cluster scalability, therefore, is to reduce this communication overhead.

The communication overhead can be reduced in two ways, (1) by reducing the size of the messages sent to the dispatcher, and (2) by limiting the number of network packets sent to the dispatcher. For (1), the distributor *compacts* the request URL by hashing it into a 32-bit integer before sending it to the dispatcher. (2) is achieved by *batching* together several messages before they are sent to the dispatcher. This enables several messages to be put in the same TCP packet and reduces interrupt and protocol processing overheads at the dispatcher node.

An interesting design choice is the degree of batching, i.e., the number of messages collected together before sending them to the dispatcher. Although the overhead on the dispatcher node is lower with a higher degree of batching, the response time is adversely affected because messages might be held for a long time before a batch of the requisite size is formed.

Rather than fixing the degree of batching, our implementation batches messages only as long as the replies for messages in the previous batch are outstanding. This has the beneficial effect of dynamically adjusting the degree of batching based on the load on the dispatcher node. If the dispatcher is heavily loaded, the time taken by it to respond to messages in an earlier batch is long, which in turn causes high degree of batching of the next set of messages. Similarly, a lightly loaded dispatcher would result in sets of messages where the degree of batch-

ing is low.

# 6 Experimental Results

In this section, we present performance results obtained with our prototype cluster. Section 6 presents performance results of the software layer-4 switch discussed in Section 5.2. In Section 6.1, we present performance characteristics of our cluster prototype such as scalability, throughput and latency. Finally, Section 6.2 provides experimental results on a real workload obtained from webserver logs. For all cluster experiments we used the configuration shown in Figure 7. The experimental setup used was the same as described in Section 3.

The first experiment determines the maximum throughput of our software layer-4 switch used as a front-end. The switch receives packets from seven client machines. The packet processing in the switch closely emulates the processing in actual operation with the cluster. Each packet was forwarded to another machine where it was discarded.

Our results show that the peak throughput afforded by the switch running on a 300MHz Pentium II machine is about 128,000 packets/s. In actual cluster operation, we have observed an average of about 5–6 packets for a typical HTTP/1.0 connection where the content size ranges from 5–13 KB. This implies that the maximum throughput afforded by the switch is about 20,000 conn/s.

Higher switch performance can be obtained by (1) using faster hardware for the switch, (2) using an SMP based machine for the switch, or (3) by using a commercial layer-4 switch. Hardware implementations of layer-4 switches are reported to achieve throughputs of over 70,000 conn/s [1].

## 6.1 Cluster Results

We repeated the experiments from Section 3 to demonstrate the scalability of our proposed cluster configuration. Figure 9 shows the throughput results with the Apache webserver as the number of nodes in the cluster (other than the one running the dispatcher) are increased. As shown in the figure, the throughput increases linearly with the size of

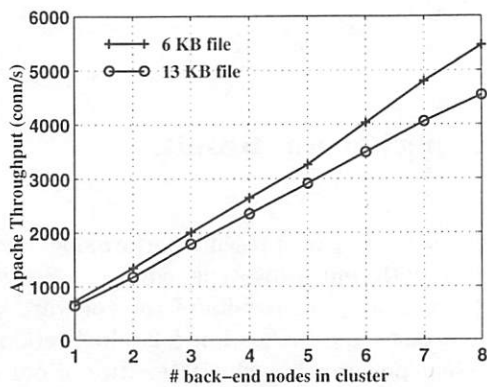


Figure 9: Cluster Throughput

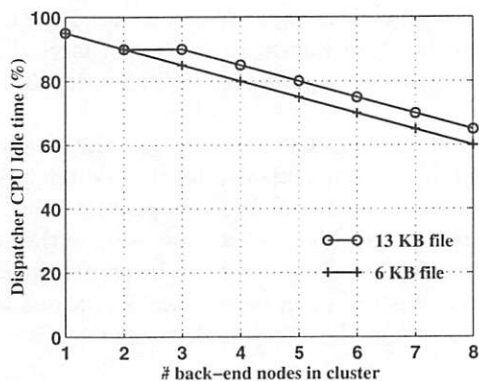


Figure 10: Dispatcher CPU Idle Time

the cluster. Figure 10 depicts the CPU idle time on the node running the dispatcher. The results clearly show that in this experiment, the dispatcher node is far from causing a bottleneck in the system.

A comparison with earlier results in Figure 2 and Figure 3 shows that the absolute performance in Figure 9 is less for comparable number of nodes in the cluster. In fact, the performance with  $N$  nodes in Figure 9 was achieved with  $N - 1$  back-end nodes in our earlier results with the conventional content-based request distribution based on handoff. The reason is that with the former approach, the front-end offloads the cluster nodes by performing the request distribution task, leaving more resources on the back-end nodes for request processing. However, this very fact causes the front-end to become a bottleneck, while our proposed approach scales with the number of back-end nodes.

Due to a lack of network bandwidth in our PIII cluster, we were unable to extend the above experiment so as to demonstrate when the dispatcher node be-

comes a bottleneck. However, we devised a different experiment to indirectly measure the peak throughput afforded by our system. We wrote a small program that generated a high rate of requests for the dispatcher node. Requests generated by this program appeared to the dispatcher as if they were requests from distributor nodes from a live cluster. This program was parameterized such that we were able to vary the degree of batching in each message sent to the dispatcher. Two messages to the dispatcher are normally required per HTTP request—one to ask for a server node assignment and one to inform the dispatcher that the client connection was closed. The degree of batching determines how many of these messages are combined into one network packet sent to the dispatcher.

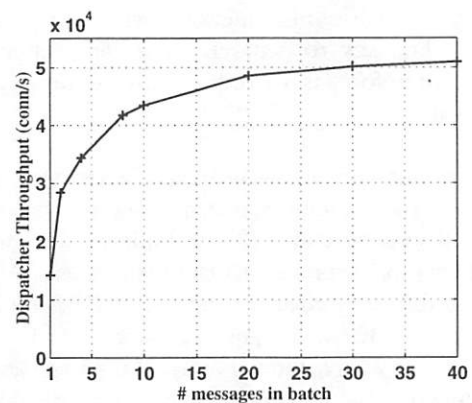


Figure 11: Dispatcher Throughput

Figure 11 shows the peak throughput afforded by the dispatcher as the degree of batching was increased from one to forty. These results show that the dispatcher node (300MHz PII) can afford a peak throughput of more than 50,000 conn/s. This number determines the peak performance afforded by our cluster and is an order of magnitude larger than that afforded by traditional clusters that employ content-aware request distribution. At this peak throughput, each HTTP connection imposes an overhead of about 20  $\mu$ sec on the dispatcher node. The bulk of this overhead is attributed to the overheads associated with communication. The LARD request distribution strategy only accounts for 0.8  $\mu$ sec of the overhead. Repeating this experiment using a 500MHz PIII PC as the dispatcher resulted in a peak throughput of 112,000 conn/s. Also, it is to be noted that the peak performance of the dispatcher is independent of the size of the content requested. In contrast, the scalability of cluster configurations with conventional content-aware request distribution decreases as the average content



size increases.

We also measured the increase in latency due to the extra communication incurred as a result of the dispatcher being placed on a separate node. This extra hop causes an average increase of 170  $\mu$ sec in latency and is largely due to round trip times in a LAN and protocol processing delays. When the layer-4 switch is used for interfacing with the clients, an additional 8  $\mu$ sec latency is added due to packet processing delay in the switch. This increase in latency is insignificant in the Internet where WAN delays are usually larger than 50 ms. Even in LAN environments, the added latency is not likely to affect user-perceived response times.

## 6.2 Real Workload

We now present results from our prototype to demonstrate the scalability of the cluster when presented with real, trace-based workloads.

The workloads were obtained from logs of a Rice University Web site and from the www.ibm.com server, IBM's main Web site. The data set for the Rice trace consists of 31,000 targets covering 1.015 GB of space. This trace needs 526/619/745 MB of cache memory to cover 97/98/99% of all requests, respectively. The data set for the IBM trace consists of 38,500 targets covering 0.984 GB of space. However, this trace has a much smaller working set and needs only 43/69/165 MB of cache memory to cover 97/98/99% of all requests, respectively.

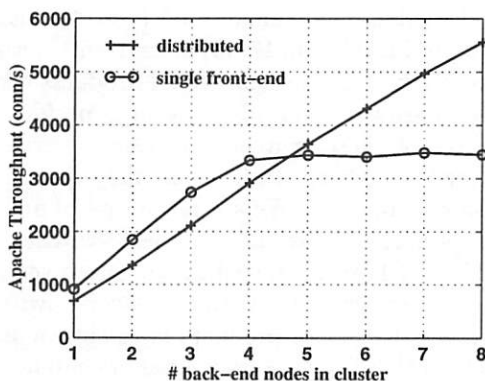


Figure 12: Throughput, IBM Trace

The results presented in Figures 12 and 13 clearly show that our proposed cluster architecture for content-aware request distribution scales far better

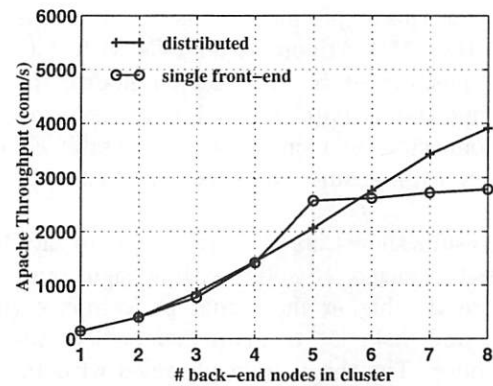


Figure 13: Throughput, Rice Trace

than the state-of-the-art approach based on handoff at a single front-end node, on real workloads.

Note that for cluster sizes below five on the IBM trace, the performance with a single front-end node exceeds that of our distributed approach. The reason is the same as that noted earlier: with the former approach, the front-end offloads the cluster nodes by performing the request distribution task, leaving more resources on the back-end nodes for request processing. However, this very fact causes the front-end to bottleneck at a cluster size of five and above, while the distributed approach scales with the number of back-end nodes.

This effect is less pronounced in the Rice trace, owing to the much larger working set size in this trace, which renders disk bandwidth the limiting resource. For the same reason, the absolute throughput numbers are significantly lower with this workload.

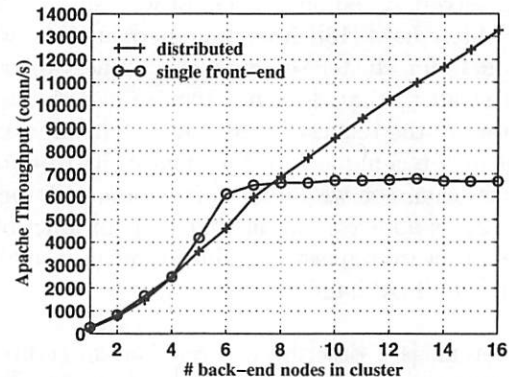


Figure 14: Throughput, Rice Trace on 800MHz Athlon

Finally, Figure 14 shows results obtained with the Rice trace on a new, larger cluster. The hardware

used for this experiment consists of a cluster of 800MHz AMD Athlon based PCs with 256MB of main memory each. The cluster nodes, the node running the dispatcher, and the client machines are connected by a single 24 port Gigabit Ethernet switch. All machines run FreeBSD 2.2.6.

The results show that the performance of the cluster scales to a size of 16 nodes with no signs of slowing, despite the higher individual performance (faster CPU and disk, larger main memory) of the cluster nodes. The throughput obtained with 16 nodes exceeds 13,000 conn/s on this platform, while the single (800MHz) front-end based approach is limited to under 7,000 conn/s. Limited hardware resources (i.e., lack of a Gigabit Ethernet switch with more than 24 ports) still prevent us from demonstrating the scalability limits of our approach, but the measured utilization of the dispatcher node is consistent with our prediction that the approach should scale to at least 50,000 conn/s.

## 7 Related Work

A substantial body of work addresses the design of high-performance, scalable Web server clusters. The work includes cooperative caching proxies inside the network, push-based document distribution and other innovative techniques [7, 10, 13, 22, 23, 28]. Our proposal addresses the complementary issue of providing support for scalable network servers that perform content-based request distribution.

Web servers based on clusters of workstations/PCs are widely used [18]. Most commercial Web switch products for cluster servers use a request distribution strategy that does not require examining the content of the request [2, 20, 14, 9]. The most common such technique used for request distribution is some variant of weighted round-robin. Resonate, Inc. [27] is an exception in that their product offers content-aware request distribution using a method similar to TCP handoff.

Fox et al. [18] describe a layered architecture for building cluster-based network services. The architecture has a centralized load manager and several front-end and back-end nodes. This architecture is similar to the one shown in Figure 6; however, the request distribution strategy and the mechanisms employed are purely load-based and do not consider

the content of the requests. Our work focuses on scalable cluster configurations for content-aware request distribution.

In [26], Pai et al. explore the use of content-based request distribution in a cluster Web server environment. This work presents an instance of a content-aware request distribution strategy, called LARD. The strategy achieves both locality, in order to increase hit rates in the Web servers' memory caches, and load balancing. Performance results with the LARD algorithm show substantial performance gains over WRR.

More recently, in [31], Zhang et al. explore another content-based request distribution algorithm that looks at static and dynamic content and also focuses on cache affinity. They confirmed the results of [26] by showing that focusing on locality can lead to significant improvements in cluster throughput.

The key to content-based request distribution is that a client's request is first inspected before a decision is made about which server node should handle the request. The difficulty lies therein, that in order to inspect a request, the client must first establish a connection with a node that will ultimately not handle the request. There are currently two known viable techniques that can be used to handle this situation. They are TCP splicing [15, 12, 29], and TCP handoff [6, 26, 19]. Our proposed approach offers a third alternative that scales well with the number of back-end nodes.

As mentioned earlier, the switch component of our cluster could easily be replaced by a commercial layer-4 switch. A number of layer-4+ network switch products [1, 16, 17, 11] are currently available on the market. These commercial products use specialized hardware and advertise high performance. A subset of these switches are also advertised to be layer-7 switches, which means they can perform URL-aware routing. We are not aware of any published performance results for these switches when used for URL-aware distribution. However, since software processing is involved in layer-7 switching, we expect that these products have similar limitations to scalability as software based content-aware front-ends when used for this purpose.

## 8 Conclusions

We have presented a new, scalable architecture for content-aware request distribution in Web server clusters. Content-aware distribution improves server performance by allowing partitioned secondary storage, specialized server nodes, and request distribution strategies that optimize for locality.

Our architecture employs a level-4 switch that acts as the central point of contact for the server on the Internet, and distributes the incoming requests to a number of back-ends. In particular, the switch does *not* perform any content-aware distribution. This function is performed by *each* of the back-ends, who may forward the incoming request to another back-end based on the requested content. In order to make their request distribution decisions, the back-ends access a dispatcher node that implements the request distribution policy.

In terms of scalability, the proposed architecture compares favorably with existing approaches, where a front-end node performs content-aware request distribution. In our architecture, the expensive operations of TCP connection establishment and handoff are distributed over all back-end nodes, rather than being centralized in the front-end node. Only a minimal additional latency penalty is paid for much improved scalability. Furthermore, the dispatcher module is so fast that centralizing it on a single 300MHz PIII machine scales to throughput rates of up to 50,000 conns/sec.

We have implemented this new architecture, and we demonstrate its scalability by comparing it to a system that performs content-aware distribution in the front-end, both under synthetic and trace-driven workloads.

## 9 Acknowledgments

We are grateful to the anonymous reviewers and our shepherd, Liviu Iftode, for their helpful comments. Thanks to Erich Nahum at IBM T.J. Watson for providing us with the www.ibm.com trace. This work was supported in part by NSF Grant CCR-9803673, by Texas TATP Grant 003604, by an IBM Partnership Award, and by equipment donations

from Compaq Western Research Lab and HP Labs.

## References

- [1] Alteon WebSystems. ACEDirector. <http://www.alteonwebsystems.com>.
- [2] D. Andresen et al. SWEB: Towards a Scalable WWW Server on MultiComputers. In *Proceedings of the 10th International Parallel Processing Symposium*, Honolulu, HI, Apr. 1996.
- [3] Apache. <http://www.apache.org/>.
- [4] M. F. Arlitt and C. L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the ACM SIGMETRICS '96 Conference*, Philadelphia, PA, Apr. 1996.
- [5] M. Aron and P. Druschel. Soft timers: efficient microsecond software timer support for network processing. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, Dec. 1999.
- [6] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient Support for P-HTTP in Cluster-based Web Servers. In *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, CA, June 1999.
- [7] G. Banga, F. Douglass, and M. Rabinovich. Optimistic Deltas for WWW Latency Reduction. In *Proceedings of the 1997 USENIX Technical Conference*, Berkeley, CA, Jan. 1997.
- [8] G. Banga and P. Druschel. Measuring the capacity of a Web server under realistic loads. *World Wide Web Journal (Special Issue on World Wide Web Characterization and Performance Evaluation)*, 2(1), May 1999.
- [9] A. Bestavros, M. Crovella, J. Liu, and D. Martin. Distributed Packet Rewriting and its Application to Scalable Server Architectures. In *Proceedings of the 6th International Conference on Network Protocols*, Austin, TX, Oct. 1998.
- [10] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 USENIX Technical Conference*, San Diego, CA, Jan. 1996.

- [11] Cisco Systems Inc. LocalDirector. <http://www.cisco.com>.
- [12] A. Cohen, S. Rangarajan, and H. Slye. On the Performance of TCP Splicing for URL-Aware Redirection. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, Oct. 1999.
- [13] P. Danzig, R. Hall, and M. Schwartz. A case for caching file objects inside internetworks. In *Proceedings of the SIGCOMM '93 Conference*, San Francisco, CA, Sept. 1993.
- [14] D. M. Dias, W. Kish, R. Mukherjee, and R. Tewari. A Scalable Highly Available Web Server. In *Proceedings of the IEEE International Computer Conference*, San Jose, CA, Feb. 1996.
- [15] K. Fall and J. Pasquale. Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability. In *Proceedings of the Winter 1993 USENIX Conference*, San Diego, CA, Jan. 1993.
- [16] FORE Systems, Inc. ESX-2400/4800. <http://www.fore.com>.
- [17] Foundry Networks. ServerIron. <http://www.foundrynet.com>.
- [18] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, San Malo, France, Oct. 1997.
- [19] G. Hunt, E. Nahum, and J. Tracey. Enabling content-based load distribution for scalable services. Technical report, IBM T.J. Watson Research Center, May 1997.
- [20] G. D. H. Hunt, G. S. Goldszmidt, R. P. King, and R. Mukherjee. Network Dispatcher: A Connection Router for Scalable Internet Services. In *Proceedings of the 7th International World Wide Web Conference*, Brisbane, Australia, Apr. 1998.
- [21] IBM Corporation. IBM interactive network dispatcher. <http://www.ibm.com/software/-network/dispatcher>.
- [22] T. M. Kroeger, D. D. Long, and J. C. Mogul. Exploring the bounds of Web latency reduction from caching and prefetching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, Dec. 1997.
- [23] G. R. Malan, F. Jahanian, and S. Subramanian. Salamander: A push-based distribution substrate for Internet applications. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, Dec. 1997.
- [24] J. C. Mogul. The Case for Persistent-Connection HTTP. In *Proceedings of the ACM SIGCOMM '95 Symposium*, Cambridge, MA, 1995.
- [25] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217-252, Aug. 1997.
- [26] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.
- [27] Resonate Inc. Resonate dispatch. <http://www.resonateinc.com>.
- [28] M. Seltzer and J. Gwertzman. The Case for Geographical Pushcaching. In *Proceedings of the 1995 Workshop on Hot Topic in Operating Systems (HotOS-V)*, Orcas Island, WA, 1995.
- [29] C.-S. Yang and M.-Y. Luo. Efficient Support for Content-Based Routing in Web Server Clusters. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, Oct. 1999.
- [30] Zeus. <http://www.zeus.co.uk/>.
- [31] X. Zhang, M. Barrientos, J. B. Chen, and M. Seltzer. HACC: An Architecture for Cluster-Based Web Servers. In *Proceedings of the 3rd USENIX Windows NT Symposium*, Seattle, WA, July 1999.



# Isolation with Flexibility: A Resource Management Framework for Central Servers

David G. Sullivan, Margo I. Seltzer  
*Division of Engineering and Applied Sciences*  
*Harvard University, Cambridge, MA 02138*  
{sullivan,margo}@eecs.harvard.edu

## Abstract

Proportional-share resource management is becoming increasingly important in today's computing environments. In particular, the growing use of the computational resources of central service providers argues for a proportional-share approach that allows resource principals to obtain allocations that reflect their relative importance. In such environments, resource principals must be isolated from one another to prevent the activities of one principal from impinging on the resource rights of others. However, such isolation limits the flexibility with which resource allocations can be modified to reflect the actual needs of applications. We present extensions to the lottery-scheduling resource management framework that increase its flexibility while preserving its ability to provide secure isolation. To demonstrate how this extended framework safely overcomes the limits imposed by existing proportional-share schemes, we have implemented a prototype system that uses the framework to manage CPU time, physical memory, and disk bandwidth. We present the results of experiments that evaluate the prototype, and we show that our framework has the potential to enable server applications to achieve significant gains in performance.

## 1 Introduction

In managing computational resources, an operating system must balance a variety of goals, including maximizing resource utilization, minimizing latency, and providing fairness. The relative importance of these goals for a particular system depends on the nature of the system and the ways in which it is used. For supercomputers running compute-intensive applications, the primary goal may be to maximize throughput, while for personal computers used to enhance a single user's productivity, the chief goal may be to maximize responsiveness.

In today's computing environments, users increasingly compete for the resources of server systems, whether to access central databases or to view content on virtually-hosted Web sites. On such systems, fairness becomes a critical resource-management goal. Proportional-share mechanisms allow this goal to be met by

providing resource principals (users, applications, threads, etc.) with guaranteed resource rights. For example, customers who pay Internet service providers to virtually host their Web sites can be given rights to shares of the hosting machine that are commensurate with the prices they pay. Service providers who can make such guarantees can offer larger resource shares to principals willing to pay a premium for better quality of service.

Although its full promise is yet to be realized, thin-client computing is another domain in which proportional-share resource management is desirable. Administrators of such systems are often forced to host one application per server to provide predictable levels of service [Sun98]. Proportional-share techniques enable the consolidation of multiple applications onto a single server by giving each application a dedicated share of the machine.

A system that supports proportional-share resource management must *isolate* resource principals from each other, so that a given principal's resource rights are protected from the activities of other principals. To provide such isolation, a system must necessarily impose limits on the flexibility with which resource allocations can be modified. Such limits work well when the resource needs of applications are well-known and unchanging, because a system administrator can assign the appropriate resource shares and leave the system to run. Unfortunately, these conditions frequently do not hold. Even if the applications' current resource needs are adequately understood, they will typically change over time. For example, as a Web site's working set of frequently accessed documents expands, the site may require an increasing share of the server's disk bandwidth in order to offer reasonable responsiveness. Moreover, it would be preferable if system administrators could be freed from the need to make detailed characterizations of applications' resource needs. Ideally, the applications themselves should be able to modify their own resource rights in response to their needs and the current state of the system.

In this paper, we present extensions to the lottery-scheduling resource management framework [Wal94, Wal95, Wal96] that allow resource principals to

safely overcome the limits on flexible allocation that proportional-share frameworks impose for the sake of secure isolation. Our extended framework supports both absolute resource reservations (*hard shares*) and proportional allocations that change in size as resource principals enter and leave the competition for a resource (*soft shares*). It also introduces a system of access controls to protect the isolation properties that lottery scheduling provides. And our framework offers the means for applications to modify their own resource rights without compromising the rights of other resource principals. One of these mechanisms, called *ticket exchanges*, allows applications to coordinate their use of the system's resources by bartering over resource rights with each other. Our extended framework thereby provides *isolation with increased flexibility*: the flexibility to safely overcome the limits on resource allocation that standard proportional-share frameworks enforce.

We have developed a prototype implementation of our framework in the VINO operating system [Sel96] and have used it, in conjunction with several proportional-share mechanisms, to manage CPU time, physical memory, and disk bandwidth. Our experiments demonstrate that the extended lottery-scheduling framework enables server applications to achieve improved performance under realistic usage scenarios.

This work makes several contributions. First, we extend the lottery-scheduling framework to securely manage multiple resources, providing both soft and hard resource shares. To our knowledge, our prototype is the first implementation of a proportional-share framework to support both types of shares for multiple resources. Second, we point out an important tension between the conflicting goals of secure isolation and flexible resource allocation, and we present mechanisms that allow for more flexible allocation while preserving secure isolation. Third, we illustrate the value of a system that can support dynamic adjustments to the resource allocations that applications receive.

In the next section, we review the original lottery-scheduling framework and describe how we extend it to securely support proportional sharing of multiple resources. In Section 3, we illustrate how lottery scheduling (like all proportional-share schemes) imposes both upper and lower limits on the resource allocations that clients can obtain, and we describe the mechanisms that we use to overcome both sets of limits while maintaining secure isolation. In Section 4, we describe our prototype implementation of the extended framework, including the scheduling mechanisms that we have chosen to employ. Section 5 presents experiments designed to evaluate the prototype and to test one of our mechanisms for flexibly adjusting resource rights. Finally, we discuss related work and summarize our conclusions.

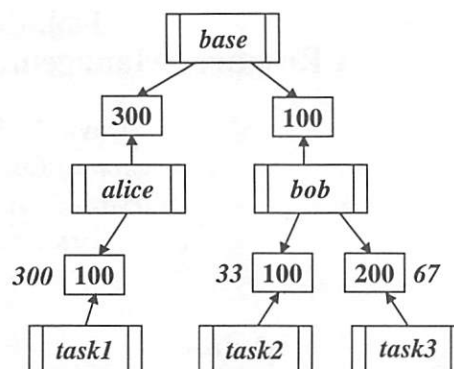


Figure 1. A sample resource hierarchy in which currencies provide isolation between the tasks of different users. The base values of the tasks' backing tickets are shown in italics.

## 2 Securely Managing Multiple Resources

### 2.1 The Original Framework

The resource management framework developed for lottery scheduling [Wal94, Wal95, Wal96] is based on two key abstractions, *tickets* and *currencies*. Tickets are used to encapsulate resource rights. Resource principals receive resource rights that are proportional to the number of tickets that they hold for a resource; changing the number of tickets held by a resource principal automatically leads to a change in its resource rights.

Tickets are issued by currencies, which allow resource principals to be grouped together and isolated from each other. Principals funded by a currency share the resource rights allotted to that currency; currencies thus enable hierarchical resource management.

Each currency effectively maintains its own exchange rate with a central *base currency*, and tickets from different currencies can be compared by determining their value with respect to the base currency (their *base value*). The more tickets a currency issues, the less each ticket is worth with respect to the base currency, and their total base value can never exceed the value of the tickets used to back the currency itself.

The sample currency hierarchy shown in Figure 1 illustrates these concepts. The *bob* currency is funded by 100 of the 400 base-currency tickets, and it thus receives rights to one-quarter of the resource. These rights are divided up by the tasks funded by *bob*; for example, *task3* holds 200 of the 300 *bob* tickets, and it thus receives rights to two-thirds of *bob*'s quarter share, or one-sixth of the total resource rights. In other words, *task3*'s 200 *bob* tickets have a base value of approximately 67 (two thirds of 100). If *task2* or *task3* forks off more tasks, causing the *bob* currency to issue more tickets, the value of its tickets will decrease, because its resource rights will be shared by a larger number of tasks. However, the resource rights of processes funded

by other currencies will not be affected.

While a lottery-scheduling resource hierarchy typically has a tree-shaped structure like the one shown in Figure 1, it can more generally take the form of any directed acyclic graph. The lottery-scheduling framework thus supports a greater variety of configurations than most other, recently proposed schemes for hierarchical resource management (see Section 6). For example, on a system like the one depicted in Figure 1, in which each user's applications are funded by a currency specific to that user, two or more users could pool their resources to support a single application that all of them are using (the system developed by Banga et al. [Ban99] also allows this).

## 2.2 Resource-Specific Tickets

Although prior implementations of lottery scheduling have focused exclusively on single resources (primarily the CPU), the original lottery-scheduling framework was designed to support multiple resources. Waldspurger [Wal95] considered two approaches to implementing a multi-resource system. In the first approach, tickets can be applied to any resource, allowing resource principals to shift tickets from one resource to another as needed, while in the second, tickets are resource-specific. Waldspurger favored the former approach because of its greater flexibility and simplicity. However, allowing principals to devote tickets to resources as they see fit violates the insulation properties of currencies, because it can lead to changes in the total number of tickets applied toward a given resource [Sul99a].

We therefore chose to use resource-specific tickets. To avoid the overhead of maintaining a separate currency configuration for each resource, we extend currencies to encompass all of the resources being managed. Concretely, this means that most pieces of currency state are maintained as arrays indexed by resource type. Similarly, many currency-related operations take a parameter that specifies the resource type.

## 2.3 Currency Brokers

For the lottery-scheduling framework to be secure in a multi-user setting, a system of access controls are needed. We encapsulate these controls in a *broker* associated with each currency. A broker stores the owner and group of the user who created the currency, along with a UNIX-style mode specifying who may perform various operations on the currency. Before these operations are carried out, the broker verifies that the current thread belongs to a user with the requisite permissions.

Like UNIX file modes, currency modes include three sets of permissions: one for the currency's owner, one for the currency's group, and one for all others. In a

given set of permissions, the *f* bit indicates whether a user is allowed to fund the currency; the *c* bit indicates whether a user can "change" the currency by removing some of its funding or destroying it entirely; and the *i* bit indicates if a user is allowed to issue or revoke the currency's own tickets. This *fci* collection of bits is comparable to the *rwX* combination in UNIX file modes.

Table 1 provides more specifics about the permission checks that brokers perform. In most cases, superusers are allowed to override the ordinary permissions checks. If an attempt to fund a currency would lead to a cycle in the currency graph, the attempt is rejected.

**Table 1.** Permission checks performed by brokers

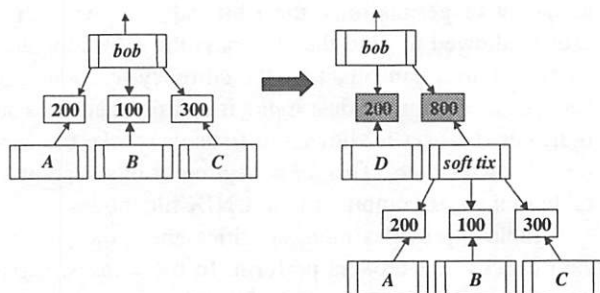
Operation	Permission check
create a currency	The caller must match both the user id and group id specified for the new currency <sup>a</sup> .
destroy a currency	The appropriate <i>c</i> ( <i>change</i> ) bit must be set in the currency's mode.
fund currency A with tickets issued by currency B	The appropriate <i>f</i> ( <i>fund</i> ) bit in A's mode <b>and</b> the appropriate <i>i</i> ( <i>issue</i> ) bit in B's mode must be set.
take tickets issued by currency B away from currency A	Either the appropriate <i>c</i> ( <i>change</i> ) bit in A's mode <b>or</b> the appropriate <i>i</i> ( <i>issue</i> ) bit in B's mode must be set.

a. Note that the user and group ids must be specified—and therefore checked—because superusers can create currencies that have ids other than their own.

## 2.4 Hard and Soft Resource Shares

The standard lottery-scheduling framework was primarily designed to support *soft* resource shares whose absolute value may change over time as principals enter and leave the competition for the resource. However, Waldspurger and Weihl pointed out that absolute, *hard* resource shares can be supported using the same framework by fixing the total number of tickets issued by the system [Wal96]. In particular, they proposed specifying hard shares by issuing tickets from a *hard currency* that maintains a fixed exchange rate with the base currency. When this hard currency issues additional tickets, some of the funding of other, "soft" currencies is transferred to the hard currency so that its exchange rate can be maintained.





**Figure 2: Offering Hard Shares of a Currency's Resource Rights.** The *bob* currency issues a hard ticket to task D representing a fixed 20% (200/1000) of *bob*'s resource rights. As a result, a special currency (*soft\_tix*) is created and used to fund *bob*'s soft tickets, isolating the hard tickets from changes in the number of soft tickets. The funding given to the *soft\_tix* currency is adjusted as needed to ensure that the total number of hard tickets issued by *bob* remains fixed at 1000.

In our framework, we take a slightly different approach based on the notion of hard and soft tickets, and we allow resource principals to obtain hard shares from any currency. Under our approach, tickets issued by a currency are ordinarily *soft tickets* that specify soft shares of the currency's resource rights. However, when a currency issues a *hard ticket* to specify a fixed percentage of its resource rights, a separate currency is created and used to fund the currency's soft tickets (Fig. 2). The number of hard tickets used to fund this soft-ticket currency is adjusted as needed to ensure that the total number of the currency's hard tickets remains fixed.<sup>1</sup>

Our approach requires no extra overhead in the common case of a currency issuing only soft tickets, and yet it still allows hard tickets to be issued by any currency. Users could use hard tickets to give an application a fixed percentage of their resource rights, or to specify hierarchical reservations in which absolute shares from the base currency are divided into hard subshares. For a hard ticket to represent a fixed-share reservation of the actual resource, all paths from the root currency to the ticket must involve only hard tickets.

### 3 Isolation with Greater Flexibility

Currencies, like all mechanisms for providing isolation, necessarily impose limits on the flexibility with which resource allocations can be modified. In the following sections, we demonstrate that currencies enforce both upper and lower limits on resource allocations. We also describe the mechanisms that we have developed to safely overcome these limits so that applications can obtain allocations that better meet their differing and dynamically changing needs.

1. Note that even the base currency's soft tickets have a base value that can change over time as the number of its hard tickets changes.

### 3.1 Problem: Currencies Impose Upper Limits

When a resource principal is funded by a currency other than the root currency, its resource rights can usually be increased by giving it extra tickets from that currency.<sup>2</sup> For example, in Figure 1, *task2*'s resource rights could be boosted by giving it 200 *bob* tickets rather than 100. However, doubling the tickets held by *task2* does not double its resource rights; rather, *task2* goes from having one-third of the *bob* currency's overall resource rights (a base value of 33) to having one-half of those rights (a base value of 50). This smaller increase reflects the fact that issuing additional *bob* tickets decreases their value. No matter how many currency tickets a resource principal receives, the resource rights imparted by those tickets cannot exceed the overall rights given to the currency itself. This upper limit is essential to providing isolation. Without it, the resource rights of principals funded by other currencies could be reduced.

Despite the need for the upper limits imposed by currencies, these limits may often be unnecessarily restrictive. This is especially true on central servers, because the large number of resource principals that a server must accommodate makes it difficult for a single allocation policy to adequately address their different and dynamically changing resource needs. Instead, some simple policy for ensuring fairness is likely to be used, such as giving users equal resource rights to divide among their applications, or allocating resource shares based on how much a user has paid.

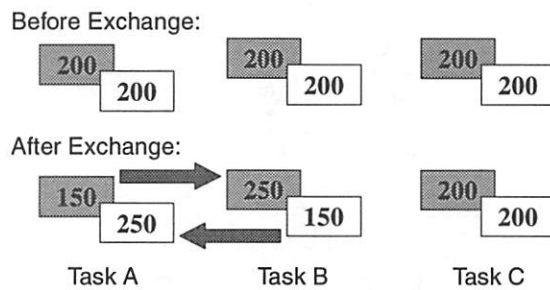
### 3.2 Solution: Ticket Exchanges

Because certain resources may be more important than others to the performance of an application, applications may benefit from giving up a fraction of their resource rights for one resource in order to receive a larger share of another resource. We have therefore developed a mechanism called *ticket exchanges* that allows applications to take advantage of their differing resource needs by bartering with each other over resource-specific tickets. For example, a CPU-intensive application could exchange some of its disk tickets for some of the CPU tickets of an I/O-intensive application.

While ticket exchanges allow principals to obtain additional resource rights, they do so without compromising the isolation properties of the lottery-scheduling framework. As the scenario depicted in Figure 3 illustrates, only the resource rights of principals participating in an exchange are affected by it; the resource rights of non-participants remain the same.

2. This is not always the case. If a resource principal is the sole recipient of a currency's tickets, giving it more tickets from the currency does not affect its resource rights.





**Figure 3: Ticket Exchanges Insulate Non-Participants.** Tasks A and B exchange tickets. Task C is unaffected, because it still has one-third of the total of each ticket type.

Ticket exchanges are not, however, guaranteed to preserve the actual resource *shares* that non-participants received before the exchange. Because the principals involved in an exchange typically make greater use of the resource for which they obtain extra tickets than the principal who traded the tickets away, resource contention will likely increase. As a result, non-participants who previously received larger resource shares than their tickets guaranteed may see those shares reduced. For example, if a CPU-intensive process trades some of its disk tickets to a process that regularly accesses the disk, those previously inactive disk tickets will suddenly become active, and the disk tickets of other processes accessing the disk may decline in value.<sup>3</sup> However, principals should always receive at least the minimal shares to which their tickets entitle them.

Ticket exchanges and currencies complement each other. Exchanges allow for greater flexibility in the face of the upper limits imposed by currencies, while currencies insulate processes from the malicious use of exchanges. For example, a process could fork off children that use exchanges to give the parent process all of their tickets. With currencies, however, this tactic would only affect the resource rights of tasks funded by the same currency as the malicious process.

### 3.2.1 Determining and Coordinating Exchanges.

Ticket exchanges enable applications to coordinate with each other in ways that are mutually beneficial and that may increase the overall efficiency of the system. Various levels of sophistication could be employed by applications to determine what types of exchanges they are willing to make and at what rates of exchange.

Certain types of resource principals may primarily need extra tickets for one particular resource. For exam-

3. When a resource principal temporarily leaves the competition for a resource (e.g., when a thread is not performing I/O), its tickets are *deactivated*. As a result, the resource rights of other principals funded by the same currency or currencies are temporarily increased until the principal reenters the competition and its tickets are reactivated.

ple, consider two Web sites that are virtually hosted on the same server. Site A has a small number of frequently accessed files that it could keep in memory if it had additional memory tickets for its currency. Site B has a uniformly accessed working set that is too large to fit in memory; it would benefit from giving up some of its currency's memory tickets for some of A's disk tickets.

Applications could also apply economic and decision-theoretic models to determine, based on information about their performance (such as how often they are scheduled and how many page faults they incur) and the current state of the system, when to initiate an exchange and at what rate. This determination could be made by the application process itself, or by a separate *resource negotiator* process that monitors the relevant variables and initiates exchanges on the application's behalf. Resource negotiators are similar to the *application managers* proposed by Waldspurger [Wal95].

Applications are free to cancel exchanges in which they are involved. This allows them to take a trial-and-error approach, experimenting with exchange rates until they achieve an acceptable level of performance and adapting their resource usage over time.

Applications or their negotiators initiate exchanges by sending the appropriate information to a central *dealer*. The dealer maintains queues of outstanding exchange proposals, attempts to match up complementary requests, and carries out the resulting exchanges. If an exchange request cannot be immediately satisfied, the dealer returns a message that includes any proposals with conflicting exchange rates (e.g., process A requests 20 CPU tickets for 10 memory tickets, while process B requests 10 memory tickets for 10 CPU tickets). In this way, an application can decide whether to modify its proposed exchange rate and try again for a compromise deal. In environments where isolation is less important, the dealer could be modified to carry out exchanges that processes propose on the processes themselves (e.g., to take away 20 CPU tickets from a process and give it 20 memory tickets in return), giving an approach equivalent to the one suggested by Waldspurger (see Section 2.2).

Future research is needed to develop negotiators suitable for a wide variety of applications and environments. Among the questions that still need to be addressed are: How can a negotiator determine what exchanges are beneficial to its associated process? When should a negotiator accept a trade less desirable than the one it proposed? Will a system involving dynamic ticket exchanges be stable (i.e., how can oscillatory behavior be avoided)? Can general-purpose negotiators be written that avoid the need to craft one for each application? In addition, the central dealer must be designed to deal fairly with requests that have complementary but differing exchange rates.

**3.2.2 Carrying Out an Exchange.** Once a complementary set of exchange requests is found, the funding of the resource principals involved must be modified to reflect the exchange. In a non-hierarchical system with only a base currency, this could be accomplished by directly modifying the number of tickets that the principals hold for the resources involved. However, the presence of non-base currencies complicates matters, because the base value of their tickets can change over time, whereas tickets used in exchanges should have a constant value.

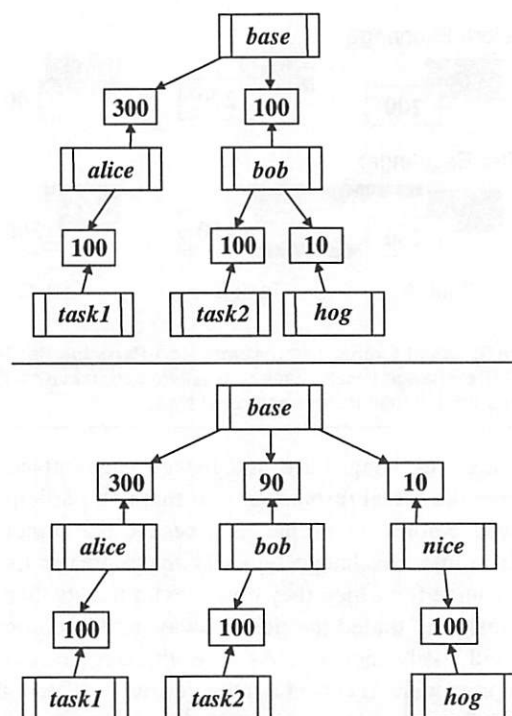
To address this problem, we issue four tickets directly from the base currency: two for the amounts being exchanged, and two *negatively valued* tickets for the opposite amounts. For example, if task A trades disk tickets with a base value of 50 for some of task B's memory tickets with a base value of 20, then A is given 20 base-currency memory tickets and -50 base-currency disk tickets, and B is given 50 base-currency disk tickets and -20 base-currency memory tickets. Because scheduling and allocation decisions are based on the total base value of a principal's backing tickets, the negative tickets reduce the principals' resource rights by the amount they have traded away. And because principals cannot manipulate their own backing tickets, exchanged tickets cannot be misused (e.g., to reduce another currency's allocations). If a principal's tickets for a resource are temporarily deactivated (see footnote 3), the funding it has obtained through exchanges for that resource is temporarily transferred to the principal's other funders to preserve isolation.

An exchange is undone if one of the exchanging principals exits, cancels the exchange, or loses the resource rights that it traded away. This last scenario can occur if the tickets funding the principal decrease in value to the point that their base value is less than the value of the tickets that the principal gave away. In such cases, the principal's total base value for that resource becomes negative, and the exchange must be revoked.

### 3.3 Problem: Currencies Impose Lower Limits

Currencies can also impose lower limits on resource rights. These limits materialize when only one of the resource principals funded by a currency is actively competing for a resource. In such circumstances, that principal receives *all* of the currency's resource rights, no matter how few tickets have been used to fund it.

As a result, currencies make it difficult for lottery scheduling to support the semantics of the *nice* utility found on conventional UNIX systems. For example, a user running a CPU-intensive job may reduce its CPU funding as a favor to other users. But if the other tasks funded by the same currency are all idle, the CPU-intensive job will still get the currency's full CPU share (Fig.



**Figure 4: Currencies Impose Lower Limits.** The user *bob* tries to lower the priority of *hog*, a CPU-intensive process, by giving it only 10 tickets (top). However, if *task2* becomes idle, *hog* will still receive all of *bob*'s resource rights. One solution is to fund *hog* directly from the base currency (bottom).

4, top). The user would presumably be allowed to decrease the number of tickets backing the user currency itself, but then other jobs funded by that currency would be adversely affected when they became runnable.

### 3.4 Solution: Limited Permission to Issue Base-Currency Tickets

While upper limits are necessary for providing isolation, lower limits are an undesirable side-effect of isolation. These limits could be overcome by funding CPU-intensive applications directly from the base currency (Fig. 4, bottom). However, for reasons of security, currency access controls (Section 2.3) would ordinarily prevent unprivileged users from issuing base-currency tickets.

To circumvent this restriction, our framework allows a user to issue base-currency tickets as long as the total value of currencies owned by that user never exceeds some upper bound. In this way, users can give up a small amount of their currencies' funding and then issue that same amount from the base currency to fund resource-intensive jobs. This approach leaves the user's total resource rights unchanged (preserving the isolation of other users), and it allows jobs to run at a reduced priority without crippling the rest of the user's applications. Section 4.6 describes how we implement this policy.

## 4 Prototype Implementation

We have implemented the extended framework in VINO-0.50 ([www.eecs.harvard.edu/~vino/vino](http://www.eecs.harvard.edu/~vino/vino)), and we use it to manage CPU time, memory, and disk bandwidth. In the following sections, we describe the key details of our implementation, including the scheduling and allocation mechanisms that we employ.

### 4.1 Threads and Currencies

A thread can be thought of as a special type of currency. Like all currencies, threads hold backing tickets, and they also issue temporary *transfer tickets* to threads on which they are waiting (see Section 4.3). By having VINO's thread class inherit from its currency class<sup>4</sup>, threads can issue and receive tickets using the same methods as non-thread currencies, and other methods (e.g., the one that recursively computes a ticket's base value) can also treat threads and currencies interchangeably. Threads also reuse currency data members to keep track of the tickets that they hold and have issued.

Currencies that are also threads are identified by the process id of the thread. All other currencies are identified by a unique currency identifier (cid).

### 4.2 Currency Configuration and Permissions

By default, the system creates one currency for each active user of the system, and it funds these *user currencies* equally from the base currency. Each user's currency in turn funds the tasks of that user. The user currencies are created by means of a function, `cid_for_client()`, that is invoked when a process changes its real user id (uid); this function uses a uid to cid mapping to determine which currency should be used to fund the process. If no mapping exists for a given user, a new currency is created and funded. In either case, the process' existing funding is revoked and replaced with funding from the appropriate user currency. Once a user's login shell is correctly funded, processes forked by that shell are funded by the same currency. More generally, child processes are funded by the issuer of the first ticket in the parent's list of backing tickets for that resource.<sup>5</sup>

Each user currency is owned by the corresponding user and has a currency mode (see Section 2.3) that allows the user to manipulate it using a set of system calls added for this purpose. A user's tasks receive equal ticket allocations by default; the user can modify these allocations and thereby alter the relative resource rights

of the tasks. Users can also create currencies and fund them with tickets from their user currency. However, they cannot increase the funding of their user currency itself, because they do not have permission to issue other currencies' tickets. Thus, each user's tasks are securely isolated from the tasks of other users, and each user has the same total resource rights.

Other currency-configuration policies could also be specified. On extensible operating systems like VINO [Sma98], superusers could safely download specialized versions of the `cid_for_client` function (which is also called when a process' real group id (gid) changes) to specify arbitrary configuration schemes based on uid and gid, as well as alternative access-control policies for currencies. Approaches that do not involve extensibility could also be employed to accommodate a more limited range of possible configurations and access controls.

### 4.3 Managing CPU Time

Our prototype uses the original lottery-scheduling algorithm [Wal94] to schedule the CPU, randomly choosing an active ticket and traversing the runnable queue to find the thread that holds the ticket. In searching for the winning thread, the system computes the base value of each thread's CPU backing tickets. We cache these base values to avoid unnecessary computation, although the cached values must be invalidated whenever a change occurs in a currency's count of active tickets (e.g., when a thread starts up, blocks, or exits).

Our prototype also uses two other features of the original lottery-scheduling framework, *compensation tickets* and *ticket transfers*. Compensation tickets are issued to threads who do not use their full quantum, temporarily inflating their resource rights to give them a higher probability of being chosen when they next become runnable. Ticket transfers occur when a thread blocks attempting to acquire a kernel mutex or to allocate memory. Because the thread is itself a currency (see Section 4.1), it issues a ticket and uses it to fund the thread on which it is waiting (the holder of the mutex or the pageout daemon), transferring its resource rights to that thread. This can reduce the time that the waiting thread spends blocked, and it prevents priority inversion from occurring. When the thread is made runnable again, its transfer tickets are revoked.

A number of deterministic algorithms, including stride scheduling [Wal95] and EEVDF [Sto96], can also be used within the lottery-scheduling framework to provide increased accuracy and lower response-time variability for interactive processes. Because our work primarily addresses ways of overcoming the limits that proportional-share frameworks impose on flexible resource allocation, the algorithms used are not crucial.

---

4. VINO is written in C++.

5. Actually, if a process holds tickets from more than one currency, its children should be funded by all of these currencies. We plan to extend our implementation to deal with this case.



#### 4.4 Managing Memory

Effective proportional-share memory management is complicated by the difficulty of determining which processes are actively competing for memory and by the undesirability of a strict partitioning of memory among processes. When scheduling the CPU, threads that are blocked are simply ignored and the values of their tickets are not counted. Our data [Sul99b] indicates that a similar approach to memory management is not effective. When the system experiences heavy memory pressure, any process that blocks, even momentarily, can lose a large number of pages to the activity of the pageout daemon, resulting in erratic paging behavior and poor throughput. The obvious alternative, namely leaving the memory tickets of all processes active at all times, is also not viable, because pages belonging to idle processes tend to remain in memory indefinitely, reducing the number of pages available to active processes and effectively partitioning the total memory of the system. We have therefore chosen to give memory guarantees only to privileged processes that explicitly request them. Other processes compete for the unreserved portion of memory, which we ensure comprises at least five percent of the memory not wired by the kernel. While this approach is limited (e.g., it can easily lead to thrashing), it allows us to experiment with the resource trade-offs that applications can make.

Processes running as root can obtain hard memory shares from the base currency. Once a currency is funded with memory tickets, resource principals with appropriate permissions can obtain soft or hard sub-shares of its allocation. As with any resource, hard shares are obtained using the `reserve()` system call and soft shares using the `fund()` system call. In the common case of obtaining a hard share from the base currency, users simply specify the size in kilobytes of the memory share they are requesting. In other cases, users either specify a number of memory tickets (for soft shares) or a percentage of the issuing currency's share (for hard shares).

To maintain guaranteed shares, we altered the behavior of VINO's pageout daemon so that pages owned by processes that have not exceeded their memory guarantee are not reclaimed. This approach does not limit processes to their memory shares, but merely ensures that they can receive at least that amount. In the absence of memory pressure, processes receive as much memory as they need. If a process holds soft memory tickets, the number of pages to which it is entitled can change dynamically as the value of its tickets changes. The pageout daemon thus needs to compute the current base value of the processes' memory tickets; cached values are used whenever possible.

#### 4.5 Managing Disk Bandwidth

Our prototype supports proportional sharing of disk bandwidth using the hierarchical YFQ algorithm [Bru99b]. YFQ approximates ideal proportional sharing by maintaining a virtual time function and per-disk queues of outstanding disk requests for each resource principal. Each of the queues has an associated *finish tag* that reflects the principal's past disk activity, its current share of the disk, and the length of the request at the head of the queue. Requests from queues with the smallest finish tags are forwarded to the device driver in small batches that can be reordered by the driver or device to achieve better aggregate throughput.

A principal's disk tickets are active whenever it has an outstanding request. To adjust to dynamic changes in the number of active tickets, the base value of a principal's disk tickets is recomputed (using cached values if possible) whenever a request reaches the head of its queue, and this value is used to compute the queue's new finish tag.

#### 4.6 Emulating Nice

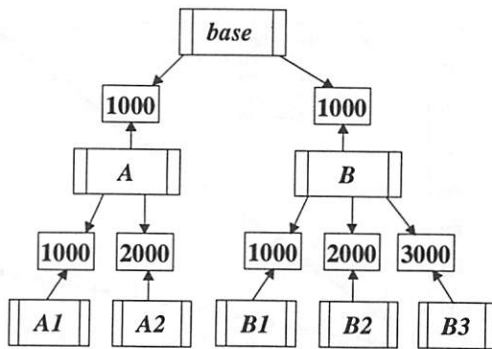
To support the semantics of *nice*, we created a utility that runs with root privileges and executes programs with funding from the base currency. This utility reduces the funding of the caller's user currency by the amount requested for the new job, thus preserving isolation. The utility actually creates a new currency for the task, funds that currency with the requested number of tickets, and uses the new currency's tickets to fund the task (see Fig. 4, *bottom*). This level of indirection is needed in case the task spawns any children; if so, they will share the funding of the new currency. While this utility would typically be used to give a small percentage of the CPU to long-running, CPU-intensive jobs, it can be used with other resources as well. It successfully overcomes the lower limits imposed by currencies without employing VINO's extensibility mechanism, as we had originally planned [Sul99a]. The other broker methods could also be overridden using similar *setuid-root* utilities.

#### 4.7 Carrying Out Exchanges

As discussed in Section 3.2.1, a number of challenging questions must be answered before a system that fully supports dynamic ticket exchanges can be built. At this point, we have implemented a framework that allows us to easily test the effects of ticket exchanges and thereby gain insight into the issues involved.

We provide two mechanisms for experimenting with exchanges. First, users with appropriate permissions can employ our `reserve()`, `fund()`, and `unfund()` system calls to implement *static* exchanges, preset modifications to the default ticket allocations.





**Figure 5.** The CPU funding used for the experiment described in Section 5.2. Currencies A and B receive equal funding from the base currency, which they divide among the tasks they fund. A2 receives twice the funding of A1, B2 receives twice the funding of B1, and B3 has three times the funding of B1.

Second, we have implemented a simple central dealer in the kernel, and we allow applications to propose exchanges dynamically using a system call (`exch_offer`) added for this purpose. When an exchange is carried out, the four tickets involved (see Section 3.2.2) are linked to each other in a circular queue so that the exchange can be invalidated when one of the principals exits, loses too much funding, or retracts the exchange. If one of the tickets is deleted, all four of them are, thereby cancelling the exchange.

## 5 Experiments

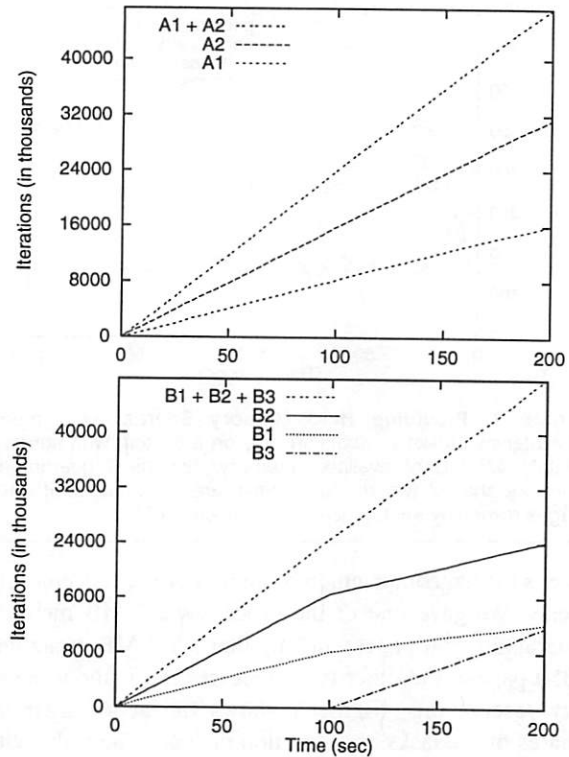
We conducted a number of experiments to test the effectiveness of our extended framework and to assess its ability to provide increased flexibility in resource allocation while preserving secure isolation. In the following sections, we first discuss tests of the proportional-share mechanisms that we implemented and demonstrate that they provide accurate proportional-share guarantees and effective isolation. We then present experiments that test the impact of ticket exchanges on two sets of applications.

### 5.1 Experimental Setup

All of these experiments were conducted using our modified kernel. We ran it on a 200-MHz Pentium Pro processor with 128 MB of RAM and a 256-KB L2 cache. The machine had an Adaptec AHA-2940 SCSI controller with a single 2.14-GB Conner CFP2105S hard disk.

### 5.2 Providing Shares of CPU Time

To test our implementation of the basic lottery-scheduling framework, we replicated an experiment from the original lottery-scheduling paper (Wal94, Section 5.5). We ran five concurrent instances of a CPU-intensive program (the dhrystone benchmark [Wei84]) for 200



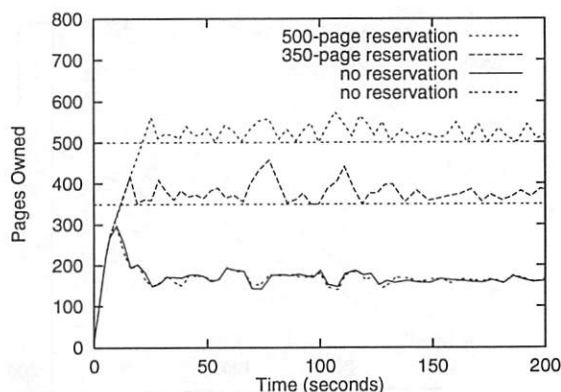
**Figure 6: Hierarchical Proportional Sharing of CPU Time.** Five CPU-intensive tasks, with funding shown in Figure 5, compete for the CPU. Shown above are the number of iterations completed by each task as a function of time. Task B3 sleeps for the first half of the test.

seconds using the CPU funding shown in Figure 5. Principal B3 sleeps for the first half of the test, during which time its tickets are not active.

Figure 6 shows the number of iterations accomplished as a function of time for the jobs funded by each currency. In all cases, the relative levels of progress of the processes match their relative funding levels. When B3 awakes, its tickets are reactivated; as a result, the other tasks funded by currency B receive reduced CPU shares, while the tasks funded by currency A are unaffected because of the isolation that currencies provide.

### 5.3 Providing Memory Shares

The next experiment tests our prototype's ability to guarantee fixed shares of physical memory. To create enough memory pressure to force frequent page reclamation, we limited the accessible memory to 8 MB. After subtracting out the pages wired by the kernel as well as the desired number of free pages in the system, there were approximately 4.2 MB of memory that principals could reserve. We ran four concurrent instances of a memory-intensive benchmark; each instance repeatedly reads random 4-KB portions of the same 16-MB file into random locations in a 4-MB buffer. This load

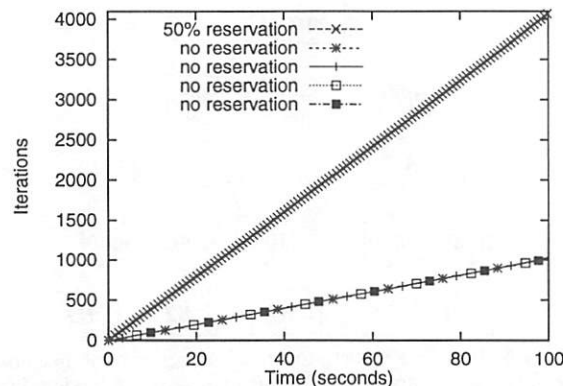


**Figure 7: Providing Hard Memory Shares.** Four memory-intensive tasks run concurrently on a system with approximately 4.2 MB of available memory. Two have guaranteed memory shares; two do not. Shown are the number of 4-KB pages owned by each process as a function of time.

keeps the pageout daemon running more or less continuously. We gave one of the processes a 2-MB memory guarantee (500 pages) and another a 1.4-MB guarantee (350 pages); the other two processes ran without memory reservations. Figure 7 shows the actual memory shares of the tasks as a function of time. The tasks with hard shares lose pages only when they own more than their guaranteed shares. The tasks without memory tickets end up owning much less memory than the ones with guaranteed shares.

### 5.4 Providing Shares of Disk Bandwidth

We tested our implementation of the YFQ algorithm for proportional-share disk scheduling by running five concurrent instances of an I/O-intensive benchmark (iohog) that maps a 16-MB file into its address space and touches the first byte of each page, causing all of the pages to be brought in from disk. Each copy of the benchmark used a different file. Throughout the test, each process almost always has one outstanding disk request. We limited YFQ's batch size to 2 for this and all of our tests to approximate strict proportional sharing. We gave one process a 50% hard share of the disk (i.e., one-half of the base currency's hard disk tickets), while the other four tasks received the default number of disk tickets from their user's currency. Figure 8 shows the number of iterations that each process accomplishes over the first 100 seconds of the test. Because one task has reserved half of the disk, the other four tasks divide up the remaining bandwidth and effectively get a one-eighth share each. Thus, the process with the hard share makes four times as much progress as the others; when it has finished touching all 4096 of its file's pages, the other four have touched approximately 1000 pages (a 4.1:1 ratio).



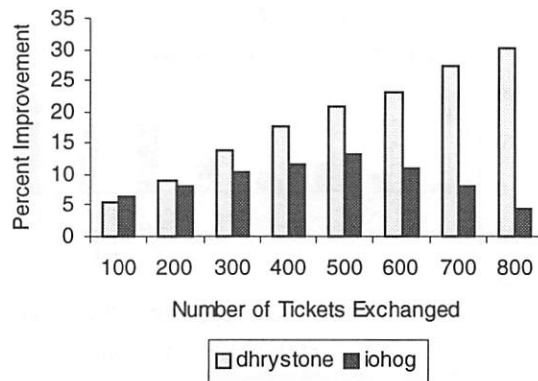
**Figure 8: Providing Proportional Shares of Disk Bandwidth.** Five I/O-intensive tasks compete for the disk. One of them receives a 50% hard share, while the others receive equal funding from their user's currency and thus divide up the other 50% of the bandwidth. Each iteration corresponds to paging in one 4-KB page of a memory-mapped file.

### 5.5 Ticket Exchanges: CPU and Disk Tickets

To study the impact of ticket exchanges, we first conducted experiments involving the CPU-intensive dhrystone benchmark [Wei84] and the I/O-intensive io hog benchmark (see Section 5.4). In the first set of runs, we gave the benchmarks allocations of 1000 CPU and 1000 disk tickets from the base currency. We then experimented with a series of one-for-one exchanges in which dhrystone gives io hog  $n$  disk tickets in exchange for  $n$  CPU tickets, where  $n = 100, 200, \dots, 800$ . To create added competition for the resources—as would typically be the case on a central server—we ran additional tasks (one dhrystone and four io hogs) in the background during each experiment. Each of the extra tasks received the standard funding of 1000 CPU and 1000 disk tickets.

Figure 9 shows the performance improvements of the exchanging applications under each exchange, in comparison to their performance under the original, equal allocations. Dhrystone benefits from all of the exchanges, and the degree of its improvement increases as it receives additional CPU tickets. Iohog also benefits from all of the exchanges, but the degree of its improvement decreases for exchanges involving more than 500 tickets. While dhrystone does almost no I/O and can thus afford to give up a large number of disk tickets, io hog needs to be scheduled in order to make progress, and thus the benefit of extra disk tickets is gradually offset by the loss of CPU tickets. However, both applications can clearly benefit from this type of exchange, which takes advantage of their differing resource needs.

We also examined the effect of the ticket exchanges on the non-exchanging tasks. As discussed in Section 3.2, the resource *rights* of these tasks should be preserved, but their actual resource *shares* may be affected.

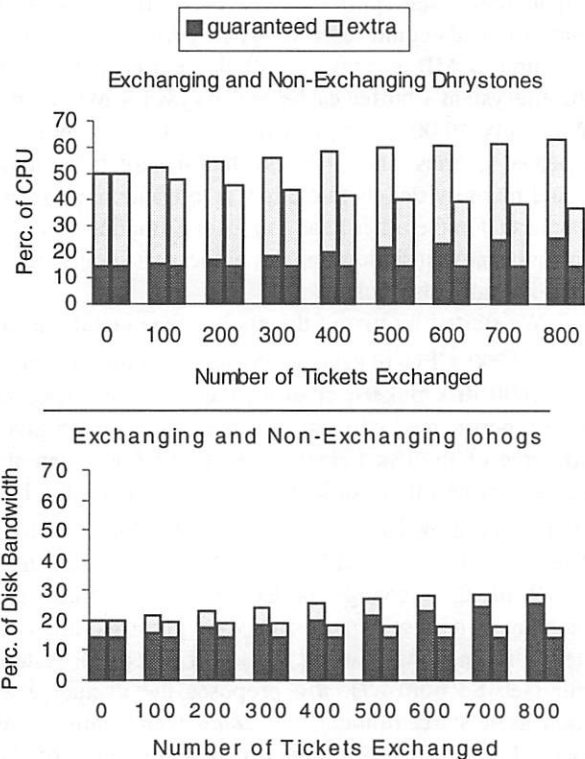


**Figure 9: Performance Improvements from Ticket Exchanges.** A CPU-intensive task (*dhrystone*) exchanges disk tickets for some of the CPU tickets of an I/O-intensive task (*iohog*). The improvements are with respect to runs in which both tasks receive the default ticket allocations. All results are averages of five runs.

Such an effect is especially likely in these experiments, because the two benchmarks rely so heavily on different resources. For example, *dhrystone* uses almost no disk bandwidth. As a result, the *iohogs* obtain more bandwidth than they would if the *dhrystones* were competing for the disk. However, when the exchanging *iohog* receives some of the exchanging *dhrystone*'s disk tickets, it obtains rights to a portion of this “extra” bandwidth, and the other *iohogs* thus end up with smaller bandwidth shares. Exchanges affect the CPU share of the non-exchanging *dhrystone* in the same way.

However, the non-exchanging processes should still obtain at least the resource shares that they would receive if all of the tasks were continuously competing for both resources. To verify this, we used `getrusage` (2) to determine each task's CPU and disk usage during the first 100 seconds of each run. The results (Fig. 10) show that the minimal resource rights of the non-exchanging processes are preserved by all of the exchanges. The top graph shows the CPU shares of both the exchanging and non-exchanging *dhrystones*, and the bottom graph shows the disk-bandwidth shares of the exchanging and non-exchanging *iohogs*.<sup>6</sup> Because there are seven tasks running during each test, the non-exchanging tasks are each guaranteed a one-seventh share (approximately 14.3%). The non-exchanging *iohogs* are affected less than the non-exchanging *dhrystone* because each of them loses only a portion of the bandwidth gained by the exchanging *iohog*. In general, as the number of tasks competing for a resource increases, the effect of exchanges on non-exchanging tasks should decrease.

6. The four non-exchanging *iohogs* have approximately equal shares. In each case, the graphed value is the smallest share of the four.



**Figure 10: Resource Shares under Exchanges.** Shown are the CPU shares of the exchanging and non-exchanging *dhrystones* (top) and the disk-bandwidth shares of the exchanging and non-exchanging *iohogs* (bottom). The dark portion of each bar represents the share guaranteed by the task's tickets, while the full bar indicates its actual share. In each pair, the left bar is the exchanging copy, and the right bar is the non-exchanging copy. All results are averages of five runs.

## 5.6 Ticket Exchanges Between Database Applications: Memory and Disk Tickets

We further experimented with ticket exchanges using two simple database applications that we developed using the Berkeley DB package [Ols99]. Both applications emulate a phone-number lookup server that takes a query and returns a number; when run in automatic mode, they repeatedly generate random queries and service them. One of the applications (*small*) has a 4-MB database with 70,000 entries, while the other (*big*) has a much larger, 64-MB database with  $2^{20}$  entries. Both applications use a memory-mapped file as a cache.

We ran these applications concurrently for a series of 300-second runs. We disabled the update thread for the sake of consistency, because its periodic flushing of dirty blocks from the applications' cache files can cause large performance variations. To emulate the environment on a busy server, we created added memory pressure—limiting the available memory to 16 MB—and we ran four *iohogs* in the background. After subtracting out the pages wired by the kernel and the system's free-page



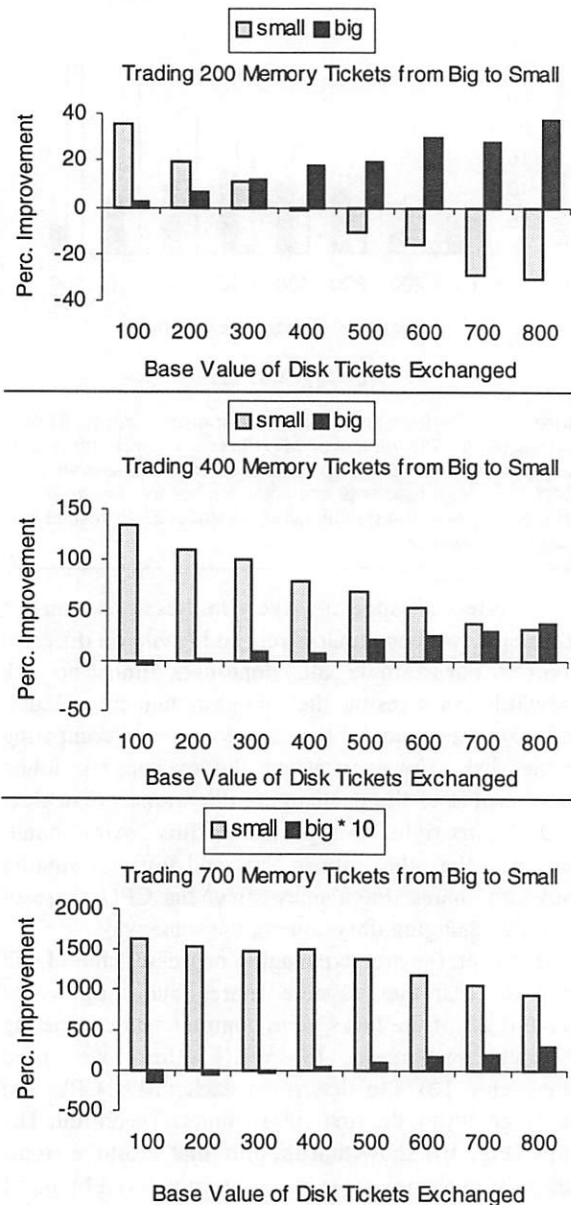
target, there was approximately 11.1 MB of memory that principals could reserve. When *small* runs alone, it uses up to 8 MB as a result of double buffering between the filesystem's buffer cache and its own 4-MB cache. With only 70,000 entries, it makes a large number of repeated queries, and it should thus benefit from additional memory tickets that allow it to cache more of its database. On the other hand, *big* uses a smaller, 500-KB cache because it seldom repeats a query; it should benefit from more disk tickets.

We started by giving the applications equal allocations: 1000 CPU tickets, 1375 hard memory tickets<sup>7</sup>, and 1000 disk tickets, all from the base currency. We then experimented with exchanges in which *small* gives up some of its disk tickets for some of *big*'s memory tickets, trying all possible pairs of values from the following set of exchange amounts: {100, 200, ..., 800}<sup>8</sup>. The *iohogs* had 1000 CPU and 1000 disk tickets each.

While the exchanges in Section 5.5 were preset, the exchanges in these experiments were proposed and carried out dynamically using the `exch_offer()` system call (see Section 4.7). *Big* proposes the exchange as soon as it starts running, but *small* waits until it has made 10,000 queries (approximately one-third of the way through the run), at which point the exchange is carried out. By waiting, *small* is able to use its original disk-ticket allocation to bring a portion of its database into memory quickly, at which point it can afford to exchange some disk tickets for memory tickets.

*Small* benefits from most of the exchanges, including any in which it obtains 400 or more memory tickets. It fails to benefit when it gains only 100 memory tickets (not shown), or when it gives away a large number of disk tickets for 300 or fewer memory tickets (Fig. 11, top). Because *small* can only fit about three-quarters of its database in memory with this allocation, it cannot afford to give away a large number of disk tickets. When *small* obtains 700 or 800 memory tickets, it can hold all of its database in memory, and it thus sees performance gains of over 1000 percent (Fig. 11, bottom). *Big* likewise benefits from most of the exchanges, including any in which it obtains disk tickets worth 600 or more.

It is interesting to note that these applications cannot simply specify an exchange *ratio*, such as two disk tickets for every one memory ticket, because what constitutes an acceptable ratio depends on the number of tickets being exchanged. For example, *small* should not accept a ratio of 2 disk for 1 memory if only 100 or 200



**Figure 11.** Results of exchanges in which an application with a large working set (*big*) exchanges memory tickets for some of the disk tickets of a similar application with a small working set (*small*). The graphed changes compare the number of requests serviced in a 100-s interval after the exchange has occurred with the requests serviced during the same interval with no exchange. Results are averages of at least five runs. There is a different vertical scale for each graph, and the values for *big* in the third graph are scaled by 10 to make them more visible. See related work [Sul99b] for graphs of the other exchanges.

memory tickets are offered, but it should accept exchanges with this ratio to obtain 300 or more memory tickets. More generally, what constitutes an acceptable exchange depends heavily on the environment in which the tasks are running. For example, because tasks need to wait until a synchronous I/O completes before

7. Each hard memory ticket from the base currency represents one page of physical memory, so 1375 tickets confer a 5.5-MB reservation.

8. Because these exchanges were carried out by the kernel, the values given represent the *base value* of the tickets exchanged, whereas the values given in Section 5.5 are the *number* of soft tickets exchanged.



enqueueing a new one, they receive at most 50% of the bandwidth in the absence of prefetching. Therefore, without extra tasks competing for the disk, *big* cannot benefit from extra disk tickets, because it already obtains 50% of the disk by default. Applications like *big* will need to use negotiators that can assess the current system conditions before proposing an exchange.

## 6 Related Work

In addition to lottery scheduling, other frameworks can be used to provide proportional-share management of multiple resources. In particular, Rialto's *activities* [Jon97], Eclipse's *reservation domains* [Bru98, Bru99a], Verghese et al.'s *Software Performance Units (SPUs)* [Ver98], and Banga et al.'s *resource containers* [Ban99] function similarly to currencies in their ability to isolate resource principals from each other.

Reservation domains and resource containers also share lottery scheduling's ability to support hierarchical resource management. However, the hierarchies supported by reservation domains are limited to a tree-shaped structure in which the resource shares of non-leaf domains are divided among their children. As discussed in Section 2.1, lottery scheduling allows resource principals to be funded by more than one currency and to thus share the resource rights of multiple currencies. Resource containers similarly allow threads to be multiplexed over several containers and to receive their combined allocations.

Moreover, most of these alternative frameworks only support hard shares; resource principals that lack a reservation either share the remaining CPU capacity equally (as in Rialto and Eclipse) or are scheduled according to a traditional time-sharing scheduling discipline. Lottery scheduling, on the other hand, can support both hard and soft shares. In their prototype implementation, resource containers were used with both fixed-share CPU guarantees and time-sharing, but they could potentially be used to support soft proportional-share guarantees as well.

The alternative approaches do provide advantages over our lottery-scheduling framework. In particular, activities and resource containers offer finer-grained resource management, addressing applications such as Web servers in which a single thread is associated with more than one independent activity. In addition, resource containers account for kernel-mode processing done on behalf of an activity. We plan to extend our lottery-scheduling framework to support these features.

Regardless of the framework used to provide proportional-share resource management, the need to isolate resource principals from each other necessarily involves imposing limits on allocations of the types

described in Sections 3.1 and 3.3. Principals restricted to a particular activity, reservation domain, SPU, or resource container cannot obtain more than their group's overall resource rights. If only one principal in a group is actively competing for a reserved resource, it will receive the entire reservation, even if it would be preferable for it to receive less than that amount. Mechanisms like ticket exchanges would be needed to allow these frameworks to provide more flexible resource allocation while preserving secure isolation.

Verghese et al.'s work on SPUs explicitly addresses the need to provide both secure isolation and flexible allocation. However, their system starts by giving absolute resource shares to each SPU, and it gains added flexibility by dividing unused portions of these shares among SPUs that need additional resources. The original lottery-scheduling framework naturally supports this type of resource sharing by deactivating the tickets of idle tasks. Our extended framework provides added flexibility through ticket exchanges and a utility that emulates the semantics of nice. One advantage of SPUs is that they were designed for use with shared-memory multiprocessors. Extending the lottery-scheduling framework for use with SMPs remains future work.

Other systems have allowed applications to negotiate their resource usage with the operating system [Jon95, Nob97]. Our extended lottery-scheduling framework lets applications coordinate their resource usage with *each other*, as well as with the system as a whole.

Besides Waldspurger's own prototypes, others have implemented portions of the lottery-scheduling framework [Arp97, Nie97]. Petrou et al. [Pet99] retrofitted lottery scheduling into FreeBSD to schedule the CPU, extending the framework to better support interactive jobs. VINO currently has a small, 10-ms quantum, so such extensions have not been needed in our prototype. Petrou et al. also suggest an alternative approach to overcoming the lower limits that currencies impose.

As discussed in Section 4.4, our scheme for managing memory is a temporary one. The Nemesis operating system [Han99] provides a more complete solution that also allows applications to obtain guaranteed memory shares. Nemesis ensures complete isolation by requiring that applications handle their own page faults.

## 7 Conclusions

Our extended lottery-scheduling resource management framework gives applications increased flexibility in modifying their resource allocations while preserving the ability to isolate groups of processes. We believe that it could be particularly useful on systems in which many users compete for the resources of a central server, as in thin-client networks or Web servers used for virtual

hosting. Ticket exchanges allow processes to adjust their allocations while insulating resource principals that do not take part in an exchange, and they enable applications to coordinate their resource usage with each other. Currency brokers provide secure access controls to currencies, while setuid utilities can be used to circumvent the default controls in ways that preserve isolation.

In order for our extended framework to be fully effective on large central servers, more work needs to be done to develop negotiators that can intelligently carry out ticket exchanges on behalf of users and applications. Developing such negotiators will be a challenging task, but one with potentially significant rewards.

## Availability

Source code and binaries for the version of VINO used in this paper, as well as source code for the test programs, can be obtained from `ftp://ftp.eecs.harvard.edu/pub/vino/vino-usenix2000`.

## Acknowledgments

This research was supported in part by a USENIX Association Scholarship. Robert Haas contributed to the development of our framework. Special thanks to Carl Waldspurger, the anonymous reviewers, and our shepherd, Yoonho Park, who all offered helpful comments on earlier drafts.

## References

- [Arp97] Arpaci-Dusseau, A.C., Culler, D.E., "Extending Proportional-Share Scheduling to a Network of Workstations," *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications*, June 1997.
- [Ban99] Banga, G., Druschel, P., Mogul, J.C., "Resource Containers: A New Facility for Resource Management in Server Systems," *Proc. of the Third Symposium on Operating Systems Design and Implementation*, February 1999.
- [Bru98] Bruno, J., Gabber, E., Özden, B., Silberschatz, A., "The Eclipse Operating System: Providing Quality of Service via Reservation Domains," *Proc. of the USENIX 1998 Annual Tech. Conference*, June 1998.
- [Bru99a] Bruno, J., Brustoloni, J., Gabber, E., Özden, B., Silberschatz, A., "Retrofitting Quality of Service into a Time-Sharing Operating System," *Proc. of the USENIX 1999 Annual Tech. Conference*, June 1999.
- [Bru99b] Bruno, J., Brustoloni, J., Gabber, E., Özden, B., Silberschatz, A., "Disk Scheduling with Quality of Service Guarantees," *Proc. of the Int'l Conf. on Multimedia Computing and Systems*, June 1999.
- [Han99] Hand, S.M., "Self-Paging in the Nemesis Operating System," *Proc. of the Third Symposium on Operating Systems Design and Implementation*, February 1999.
- [Jon95] Jones, M.B., Leach, P.J., Draves, R.P., Barrera, J.S., "Modular Real-Time Resource Management in the Rialto Operating System," *Proc. of the Fifth Workshop on Hot Topics in Operating Systems*, May 1995.
- [Jon97] Jones, M.B., Rosu, D., Rosu, M.-C., "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities," *Proc. of the 16th ACM Symposium on Operating System Principles*, October 1997.
- [Nie97] Nieh, J., Lam, M., "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications," *Proc. of the 16th ACM Symposium on Operating System Principles*, October 1997.
- [Nob97] Noble, B.D., Satyanarayanan, M., Narayanan, D., Tilton, J.E., Flinn, J., Walker, K.R., "Agile Application-Aware Adaptation for Mobility," *Proc. of the 16th ACM Symposium on Operating System Principles*, October 1997.
- [Ols99] Olson, M., Bostic, K., Seltzer, M., "Berkeley DB," *Proc. of the USENIX 1999 Annual Tech. Conference*, June 1999.
- [Pet99] Petrou, D., Milford, J., Gibson, G., "Implementing Lottery Scheduling: Matching the Specializations in Traditional Schedulers," *Proc. of the USENIX 1999 Annual Tech. Conference*, June 1999.
- [Sel96] Seltzer, M., Endo, Y., Small, C., Smith, K., "Dealing with Disaster: Surviving Misbehaved Kernel Extensions," *Proc. of the Second Symposium on Operating System Design and Implementation*, October 1996.
- [Sma98] Small, C., *Building an Extensible Operating System*, Ph.D. thesis, Division of Engineering and Applied Sciences, Harvard University, October 1998.
- [Sto96] Stoica, I., Abdel-Wahab, H., Jeffay, K., Baruah, S., Gehrke, J., Plaxton, C.G., "A Proportional-Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems," *Proc. of the IEEE Real-Time Systems Symposium*, December 1996.
- [Sul99a] Sullivan, D., Haas, R., Seltzer, M., "Tickets and Currencies Revisited: Extending Multi-Resource Lottery Scheduling," *Proc. of the Seventh Workshop on Hot Topics in Operating Systems*, March 1999.
- [Sul99b] Sullivan, D., Seltzer, M., "A Resource Management Framework for Central Servers," Computer Science Technical Report TR-13-99, Harvard University, December 1999.
- [Sun98] "Solaris Resource Manager 1.0: Controlling System Resources Effectively: A White Paper," <http://www.sun.com/software/white-papers/wp-srm/>.
- [Ver98] Verghese, B., Gupta, A., Rosenblum, M., "Performance Isolation: Sharing and Isolation in Shared Memory Multiprocessors," *Proc. of the Eighth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, October 1998.
- [Wal94] Waldspurger, C.A., Weihl, W., "Lottery Scheduling: Flexible Proportional-Share Resource Management," *Proc. of the First Symposium on Operating System Design and Implementation*, November 1994.
- [Wal95] Waldspurger, C.A., *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*, Ph.D. thesis, MIT/LCS/TR-667, MIT Laboratory for Computer Science, September 1995.
- [Wal96] Waldspurger, C.A., Weihl, W., "An Object-Oriented Framework for Modular Resource Management," *Proc. of the Fifth Int'l Workshop on Object Orientation in Operating Systems*, October 1996.
- [Wei84] Weicker, R.P., "Dhrystone: A Synthetic Systems Programming Benchmark," *Communications of the ACM*, October 1984.

# THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

## SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

## Member Benefits:

- Free subscription to *login:*, the Association's magazine, published eight-ten times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see <http://www.usenix.org/membership/specialdisc.html> for details.

## Supporting Members of the USENIX Association:

Earthlink Network	Microsoft Research	Server/Workstation Expert
Greenberg News Networks/MedCast Networks	MKS, Inc.	Sun Microsystems, Inc.
JSB Software Technologies	Motorola Australia Software Centre	Sybase, Inc.
Lucent Technologies	Nimrod AS	Syntax, Inc.
Macmillan Computer Publishing, USA	O'Reilly & Associates Inc.	UUNET Technologies, Inc.
	Performance Computing	Web Publishing, Inc.
	Sendmail, Inc.	WITSEC, Inc.

## Supporting Members of SAGE:

Collective Technologies	Mentor Graphics Corp.	RIPE NCC
Deer Run Associates	Microsoft Research	SysAdmin Magazine
Electric Lightwave, Inc.	MindSource Software Engineers	Unix Guru Universe
ESM Services, Inc.	Motorola Australia Software Centre	
GNAC, Inc.	New Riders Press	
Macmillan Computer Publishing, USA	O'Reilly & Associates Inc.	
	Remedy Corporation	

For more information about membership, conferences, or publications,  
see <http://www.usenix.org/>

or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA.  
Phone: 510-528-8649. Fax: 510-548-5738. Email: [office@usenix.org](mailto:office@usenix.org).

